

**IN THE UNITED STATES DISTRICT COURT
FOR THE WESTERN DISTRICT OF TEXAS
WACO DIVISION**

STREAMSCALE, INC.,)	
)	
Plaintiff,)	
)	
v.)	Civil No. 6:21-cv-00198-ADA
)	
CLOUDERA, INC.,)	JURY TRIAL DEMANDED
AUTOMATIC DATA PROCESSING, INC.,)	
EXPERIAN PLC, WARGAMING)	
(AUSTIN), INC., and)	
INTEL CORPORATION,)	
)	
Defendants.)	

SECONDED AMENDED COMPLAINT FOR PATENT INFRINGEMENT

Plaintiff StreamScale, Inc. (“Plaintiff” or “StreamScale”) files this Second Amended Complaint for patent infringement against Defendants Cloudera, Inc. (“Cloudera”), Automatic Data Processing, Inc.¹ (“ADP”), Experian plc (“Experian”), Wargaming (Austin), Inc. (“Wargaming”), and Intel Corporation (“Intel”) (collectively, “Defendants”) alleging as follows:

NATURE OF SUIT

1. This is a claim for patent infringement arising under the patent laws of the United States, Title 35 of the United States Code.

¹ On June 11, 2021, Defendant Automatic Data Processing, Inc. filed an unopposed motion to substitute ADP, Inc. in its place. Unopposed Motion to Substitute Party, *StreamScale, Inc. v. Cloudera, Inc.*, No. 6:21-cv-00198-ADA (W.D. Tex. June 11, 2021), ECF No. 50. To date, the Court has not yet acted on that motion. To maintain the status quo, StreamScale, Inc. has again named Automatic Data Processing, Inc. in this Second Amended Complaint, but its allegations apply equally to ADP, Inc.

2. StreamScale owns multiple patents relating to accelerated erasure coding. StreamScale's patented technology is a cornerstone of modern data storage, especially cloud-based data storage.

3. Data storage protection from loss used to be a matter of replicating the data. Data replication resulted in redundant data drives, and that redundancy provided an enhanced measure of data availability along with some measure of fault tolerance. For example, if one of the data drives were to be corrupted, the original data would still be available on a redundant disk.

4. Data replication is highly inefficient and no longer commercially practicable. Take a triple replication scheme for example. If a user desired to save some quantum of data, say 100 GB, it would require 300 GB of data storage to save that 100 GB of data. That is only a 33% utilization of storage capacity. And that measure of efficiency gets worse as the amount of redundancy in a system increases. Triple replication is also incredibly expensive because you need to buy three times the capacity of your original data. Triple replication further requires the additional, redundant capacity to be packaged, powered, and serviced.

5. Systems that employ accelerated erasure coding as patented by StreamScale enable scalable, high-performance data storage systems that can outperform other systems and do so at lower cost. StreamScale's inventions significantly reduce storage overhead while achieving similar or better fault tolerance than prior systems and methods, and are a quantum leap forward from prior systems.

6. At a high level, erasure coding uses specially designed systems to transform a block of original data to be stored into one or more blocks of encoded data that can be distributed across numerous storage devices or drives. The original data can be reconstructed from the encoded data, even if some portions of the original data are lost or unavailable. The data encoding and decoding

processes are time and energy intensive. If erasure coding is performed without appropriately configured computers using appropriately organized instructions, it can appear to have only limited practical applicability. Indeed, the widespread view in the industry before the work of StreamScale was that there was no way to employ erasure coding at high speeds, including so-called “cache line speeds.”

7. With its accelerated erasure coding technology, StreamScale achieved what was thought to be impossible. StreamScale achieved in one embodiment more than an order of magnitude performance increase in actual system performance. Rather than remaining an unobtainable goal with very limited application, storage systems based on StreamScale’s accelerated erasure coding immediately became practical and thus had newfound applicability to the data storage industry, among others.

8. The innovations described in—and protected by—StreamScale’s Patents-in-Suit have been incorporated into products and services offered by Cloudera, ADP, Experian, and Wargaming. For its part, Intel has induced infringement by at least Cloudera, ADP, Experian, and Wargaming through Intel’s collaboration with Cloudera relating to accelerated erasure coding.

PARTIES

9. Plaintiff StreamScale, Inc. is a corporation duly organized and existing under the laws of the State of Texas, having a principal place of business at 7215 Bosque Blvd., Suite 203, Waco, Texas 76710. StreamScale is the owner of record of the Patents-in-Suit in this action.

10. Defendant Cloudera, Inc. (“Cloudera”) is a corporation organized under the laws of the State of Delaware. Cloudera maintains an office in this judicial district at 515 Congress, Suite 1300, Austin, Texas 78701. Cloudera can be served with process through its registered agent in the State of Texas, Corporation Service Company d/b/a CSC – Lawyers Incorporating Service Company, 211 East 7th Street, Suite 620, Austin, Texas 78701-3218.

11. Defendant Automatic Data Processing, Inc. (“ADP”) is a corporation organized under the laws of the State of Delaware. ADP maintains offices in this judicial district, including at (i) 6500 River Place Blvd., Bldg VII, Austin, Texas 78730, (ii) 1851 North Resler, El Paso, Texas 79912, (iii) 7650 San Felipe Dr., El Paso, Texas 79912, and (iv) 211 North Loop 1604 East, San Antonio, Texas 78232. ADP can be served with process through its registered agent in the State of Texas, C T Corporation System, 1999 Bryan St., Suite 900, Dallas, Texas 75201.

12. Defendant Experian PLC is a public limited company registered and incorporated under the laws of the Bailiwick of Jersey, having a principal place of business at Newenham House, Northern Cross, Malahide Road, Dublin 17, D17 AY61, Ireland, and registered office at 22 Grenville Street, St Helier, Jersey JE4 8PX, Channel Islands.

13. Defendant Wargaming (Austin), Inc. (“Wargaming”) is a corporation organized under the laws of the State of Delaware, having a principal place of business at 11001 Lakeline Blvd., Austin, Texas 78717. Wargaming can be served with process through its registered agent in the State of Texas, C T Corporation System, 1999 Bryan St., Suite 900, Dallas, Texas 75201.

14. Defendant Intel Corporation (“Intel”) is a corporation organized under the laws of the State of Delaware. Intel maintains an office in this judicial district at 9442 N. Capital of Texas Hwy, Bldg 2, Suite 600, Austin, Texas 78759. Intel can be served with process through its registered agent in the State of Texas, C T Corporation System, 1999 Bryan St., Suite 900, Dallas, Texas 75201.

15. Collectively, Cloudera, ADP, Experian, Wargaming, and Intel are referred to herein as the “Defendants.”

JURISDICTION AND VENUE

16. This action arises under the patent laws of the United States, 35 U.S.C. § 101, *et seq.* This Court has jurisdiction over this action pursuant to 28 U.S.C. §§ 1331 and 1338(a).

17. Cloudera is subject to personal jurisdiction in this Court. This Court has personal jurisdiction over Cloudera because Cloudera has engaged in continuous, systematic, and substantial activities within this State, including substantial marketing and sales of products and services within this State and this District. Furthermore, upon information and belief, this Court has personal jurisdiction over Cloudera because Cloudera has committed acts giving rise to StreamScale's claims for patent infringement within and directed to this District.

18. Upon information and belief, Cloudera has conducted and does conduct substantial business in this forum, directly and/or through subsidiaries, agents, representatives, or intermediaries, such substantial business including but not limited to: (i) at least a portion of the acts of infringement alleged herein; (ii) purposefully and voluntarily placing one or more infringing products and services into the stream of commerce with the expectation that they will be purchased by consumers in this forum; and/or (iii) regularly doing or soliciting business, engaging in other persistent courses of conduct, or deriving substantial revenue from goods and services provided to individuals in Texas and in this judicial district. Thus, Cloudera is subject to this Court's specific and general personal jurisdiction pursuant to due process and the Texas Long Arm Statute.

19. Upon information and belief, Cloudera has committed acts of infringement in this District and has one or more regular and established places of business within this District under 28 U.S.C. § 1400(b). Thus, venue is proper in this District under 28 U.S.C. § 1400(b).

20. Cloudera maintains a permanent physical presence within this District. For example, it maintains at least the office location at 515 Congress, Suite 1300, Austin, Texas 78701. Cloudera employs numerous employees who work at Cloudera's location(s) in this District.

21. Cloudera's location(s) in this District, including at least those identified in paragraph 20 above, are regular and established places of business under 28 U.S.C. § 1391, 28 U.S.C. § 1400(b), and *In re Cray, Inc.*, 871 F.3d 1355, 1360 (Fed. Cir. 2017).

a. Cloudera's location(s) in this District, including at least those identified in paragraph 20 above, are physical, geographical location(s) in this District. Each office location comprises one or more buildings or office spaces from which the business of Cloudera is carried out. The location(s) are set apart for the purpose of carrying out Cloudera's business, including but not limited to, making, using, selling, offering for sale, and/or supporting infringing products and services. Indeed, Cloudera itself advertises its physical location(s) in this District as places of its business.

b. Cloudera's location(s) in this District, including at least those identified in paragraph 20 above, are regular and established.

c. Cloudera's location(s) in this District, including at least those identified in paragraph 20 above, are places of business of Cloudera. Cloudera conducts business from its location(s) in this District, including at least those identified in paragraph 20 above, including but not limited to, making, using, selling, offering for sale, and/or supporting infringing products and services.

d. Cloudera's location(s) in this District, including at least those identified in paragraph 20 above, are physical, geographical location(s) in this District from which Cloudera carries out its business.

e. Cloudera employees work at Cloudera's location(s), including at least those identified in paragraph 20 above. Upon information and belief, these Cloudera employees are regularly and physically present at Cloudera's location(s), including at least those

identified in paragraph 20 above, during business hours and they are conducting Cloudera's business while working there.

22. ADP is subject to personal jurisdiction in this Court. This Court has personal jurisdiction over ADP because ADP has engaged in continuous, systematic, and substantial activities within this State, including substantial marketing and sales of products and services within this State and this District. Furthermore, upon information and belief, this Court has personal jurisdiction over ADP because ADP has committed acts giving rise to StreamScale's claims for patent infringement within and directed to this District.

23. Upon information and belief, ADP has conducted and does conduct substantial business in this forum, directly and/or through subsidiaries, agents, representatives, or intermediaries, such substantial business including but not limited to: (i) at least a portion of the acts of infringement alleged herein; (ii) purposefully and voluntarily placing one or more infringing products into the stream of commerce with the expectation that they will be purchased by consumers in this forum; and/or (iii) regularly doing or soliciting business, engaging in other persistent courses of conduct, or deriving substantial revenue from goods and services provided to individuals in Texas and in this judicial district. Thus, ADP is subject to this Court's specific and general personal jurisdiction pursuant to due process and the Texas Long Arm Statute.

24. Upon information and belief, ADP has committed acts of infringement in this District and has one or more regular and established places of business within this District under 28 U.S.C. § 1400(b). Thus, venue is proper in this District under 28 U.S.C. § 1400(b).

25. ADP maintains a permanent physical presence within this District. For example, it maintains office locations at (i) ADP Austin, 6500 River Place Blvd. Bldg. VII, Austin, Texas 78730; (ii) ADP El Paso, 1851 North Resler, El Paso, Texas 79912; (iii) ADP El Paso, 7650 San

Felipe Drive, El Paso, Texas 79912; and (iv) ADP San Antonio, 211 North Loop 1604 East, San Antonio, Texas 78232. ADP employs employees who work at ADP's locations in this District.

26. ADP's location(s) in this District, including at least those identified in paragraph 25 above, are regular and established places of business under 28 U.S.C. § 1391, 28 U.S.C. § 1400(b), and *In re Cray, Inc.*, 871 F.3d 1355, 1360 (Fed. Cir. 2017).

a. ADP's location(s) in this District, including at least those identified in paragraph 25 above, are physical, geographical location(s) in this District. Each office location comprises one or more buildings or office spaces from which the business of ADP is carried out. The location(s) are set apart for the purpose carrying out ADP's business, including but not limited to, making, using, selling, offering for sale, and/or supporting infringing products and services. Indeed, ADP itself advertises its physical location(s) in this District as places of its business, and it features commercial signage at many of these location(s).

b. ADP's location(s) in this District, including at least those identified in paragraph 25 above, are regular and established. ADP features commercial signage at many of the location(s) identifying the location as a regular and established place of ADP's business.

c. ADP's location(s) in this District, including at least those identified in paragraph 25 above, are places of business of ADP. ADP conducts business from its location(s) in this District, including at least those identified in paragraph 25 above, including but not limited to making, using selling, offering for sale, and/or supporting infringing products and services.

d. ADP's location(s) in this District, including at least those identified in paragraph 25 above, are physical, geographical location(s) in this District from which ADP carries out its business.

e. ADP employees work at ADP's location(s), including at least those identified in paragraph 25 above. Upon information and belief, these ADP employees are regularly and physically present at ADP's location(s), including at least those identified in paragraph 25 above, during business hours and they are conducting ADP's business while working there.

27. Experian is subject to personal jurisdiction in this Court. This Court has personal jurisdiction over Experian because, upon information and belief, Experian has engaged in continuous, systematic, and substantial activities within this State, for example with and through its corporate subsidiaries CSIdentity Corporation and Experian Information Solutions, Inc. Upon information and belief, Experian's continuous, systematic, and substantial activities within this State include substantial marketing and sales of products and services within this State and this District, including for example through Experian's corporate subsidiaries CSIdentity Corporation and Experian Information Solutions, Inc. Furthermore, upon information and belief, this Court has personal jurisdiction over Experian because Experian has committed acts giving rise to StreamScale's claims for patent infringement within and directed to this District.

28. Upon information and belief, Experian has conducted and does conduct substantial business in this forum, directly and/or through subsidiaries, agents, representatives, or intermediaries, such substantial business including but not limited to: (i) at least a portion of the acts of infringement alleged herein; (ii) purposefully and voluntarily placing one or more infringing products into the stream of commerce with the expectation that they will be purchased

by consumers in this forum; and/or (iii) regularly doing or soliciting business, engaging in other persistent courses of conduct, or deriving substantial revenue from goods and services provided to individuals in Texas and in this judicial district. Thus, Experian is subject to this Court's specific and general personal jurisdiction pursuant to due process and the Texas Long Arm Statute.

29. To the extent Experian is not subject to jurisdiction in any State's courts of general jurisdiction, this Court has personal jurisdiction of Experian pursuant to Federal Rule of Civil Procedure 4(k)(2) because StreamScale's claims arise under federal law and exercising jurisdiction is consistent with the United States Constitution and laws.

30. Upon information and belief, Experian has committed acts of infringement in this District and has, itself or through its corporate subsidiaries, one or more regular and established places of business within this District under 28 U.S.C. § 1400(b). Thus, venue is proper in this District under 28 U.S.C. § 1400(b).

31. Experian, including for example through Experian's corporate subsidiaries CSIdentity Corporation and Experian Information Solutions, Inc., maintains a permanent physical presence within this District. For example, it maintains at least the office location at 1501 South MoPac Expressway, Austin, Texas 78746. Experian employs employees who work at Experian's location(s) in this District.

32. Experian's location(s) in this District, including at least those identified in paragraph 31 above, are regular and established places of business under 28 U.S.C. § 1391, 28 U.S.C. § 1400(b), and *In re Cray, Inc.*, 871 F.3d 1355, 1360 (Fed. Cir. 2017).

a. Experian's location(s) in this District, including for example those identified in paragraph 31 above, are physical, geographical location(s) in this District. Each office location comprises one or more buildings or office spaces from which the

business of Experian is carried out. The location(s) are set apart for the purpose of carrying out Experian's business, including but not limited to, making, using, selling, offering for sale, and/or supporting infringing products and services. Indeed, Experian itself advertises its physical location(s) in this District as places of its business, and it features commercial signage at these location(s).

b. Experian's location(s) in this District, including at least those identified in paragraph 31 above, are regular and established. Experian features commercial signage at the location(s) identifying the location as a regular and established place of Experian's business.

c. Experian's location(s) in this District, including at least those identified in paragraph 31 above, are places of business of Experian, including at least Experian's corporate subsidiaries CSIdentity Corporation and Experian Information Solutions, Inc. Experian conducts business from its location(s) in this District, including at least those identified in paragraph 31 above, including but not limited to, making, using, selling, offering for sale, and/or supporting infringing products and services.

d. Experian's location(s) in this District, including at least those identified in paragraph 31 above, are physical, geographical location(s) in this District from which Experian carries out its business.

e. Experian employees work at Experian's location(s), including at least those identified in paragraph 31 above. Upon information and belief, these Experian employees are regularly and physically present at Experian's location(s), including at least those identified in paragraph 31 above, during business hours and they are conducting Experian's business while working there.

33. To the extent Experian is found not reside in the United States, venue is nonetheless proper in this Court as to Experian pursuant to 28 U.S.C. § 1391(c)(3).

34. Wargaming is subject to personal jurisdiction in this Court. This Court has personal jurisdiction over Wargaming because Wargaming has engaged in continuous, systematic, and substantial activities within this State, including substantial marketing and sales of products and services within this State and this District. Furthermore, upon information and belief, this Court has personal jurisdiction over Wargaming because Wargaming has committed acts giving rise to StreamScale's claims for patent infringement within and directed to this District.

35. Upon information and belief, Wargaming has conducted and does conduct substantial business in this forum, directly and/or through subsidiaries, agents, representatives, or intermediaries, such substantial business including but not limited to: (i) at least a portion of the acts of infringement alleged herein; (ii) purposefully and voluntarily placing one or more infringing products and services into the stream of commerce with the expectation that they will be purchased by consumers in this forum; and/or (iii) regularly doing or soliciting business, engaging in other persistent courses of conduct, or deriving substantial revenue from goods and services provided to individuals in Texas and in this judicial district. Thus, Wargaming is subject to this Court's specific and general personal jurisdiction pursuant to due process and the Texas Long Arm Statute.

36. Upon information and belief, Wargaming has committed acts of infringement in this District and has one or more regular and established places of business within this District under 28 U.S.C. § 1400(b). Thus, venue is proper in this District under 28 U.S.C. § 1400(b).

37. Wargaming maintains a permanent physical presence within this District. For example, it maintains at least the office location at 11001 Lakeline Blvd., Austin, Texas 78717. Wargaming employs numerous employees who work at Wargaming's location(s) in this District.

38. Wargaming's location(s) in this District, including at least those identified in paragraph 37 above, are regular and established places of business under 28 U.S.C. § 1391, 28 U.S.C. § 1400(b), and *In re Cray, Inc.*, 871 F.3d 1355, 1360 (Fed. Cir. 2017).

a. Wargaming's location(s) in this District, including at least those identified in paragraph 37 above, are physical, geographical location(s) in this District. Each office location comprises one or more buildings or office spaces from which the business of Wargaming is carried out. The location(s) are set apart for the purpose of carrying out Wargaming's business, including but not limited to, making, using, selling, offering for sale, and/or supporting infringing products and services. Indeed, Wargaming itself advertises its physical location(s) in this District as places of its business.

b. Wargaming's location(s) in this District, including at least those identified in paragraph 37 above, are regular and established.

c. Wargaming's location(s) in this District, including at least those identified in paragraph 37 above, are places of business of Wargaming. Wargaming conducts business from its location(s) in this District, including at least those identified in paragraph 37 above, including but not limited to, making, using, selling, offering for sale, and/or supporting infringing products and services.

d. Wargaming's location(s) in this District, including at least those identified in paragraph 37 above, are physical, geographical location(s) in this District from which Wargaming carries out its business.

e. Wargaming employees work at Wargaming's location(s), including at least those identified in paragraph 37 above. Upon information and belief, these Wargaming employees are regularly and physically present at Wargaming's location(s), including at least those identified in paragraph 37 above, during business hours and they are conducting Wargaming's business while working there.

39. Intel is subject to personal jurisdiction in this Court. This Court has personal jurisdiction over Intel because Intel has engaged in continuous, systematic, and substantial activities within this State, including substantial marketing and sales of products and services within this State and this District. Furthermore, upon information and belief, this Court has personal jurisdiction over Intel because Intel has committed acts giving rise to StreamScale's claims for patent infringement within and directed to this District.

40. Upon information and belief, Intel has conducted and does conduct substantial business in this forum, directly and/or through subsidiaries, agents, representatives, or intermediaries, such substantial business including but not limited to: (i) at least a portion of the acts of infringement alleged herein; (ii) purposefully and voluntarily placing one or more infringing products into the stream of commerce with the expectation that they will be purchased by consumers in this forum; and/or (iii) regularly doing or soliciting business, engaging in other persistent courses of conduct, or deriving substantial revenue from goods and services provided to individuals in Texas and in this judicial district. Thus, Intel is subject to this Court's specific and general personal jurisdiction pursuant to due process and the Texas Long Arm Statute.

41. Upon information and belief, Intel has committed acts of infringement in this District and has one or more regular and established places of business within this District under 28 U.S.C. § 1400(b). Thus, venue is proper in this District under 28 U.S.C. § 1400(b).

42. Intel maintains a permanent physical presence within this District. For example, it maintains at least the office location at 9442 N. Capital of Texas Hwy, Bldg 2, Suite 600, Austin, Texas 78759. Intel employs numerous employees who work at Intel's location(s) in this District.

43. Intel's location(s) in this District, including at least those identified in paragraph 42 above, are regular and established places of business under 28 U.S.C. § 1391, 28 U.S.C. § 1400(b), and *In re Cray, Inc.*, 871 F.3d 1355, 1360 (Fed. Cir. 2017).

a. Intel's location(s) in this District, including at least those identified in paragraph 42 above, are physical, geographical location(s) in this District. Each office location comprises one or more buildings or office spaces from which the business of Intel is carried out. The location(s) are set apart for the purpose of carrying out Intel's business, including but not limited to the acts of infringement alleged herein. Indeed, Intel itself advertises its physical location(s) in this District as places of its business.

b. Intel's location(s) in this District, including at least those identified in paragraph 42 above, are regular and established.

c. Intel's location(s) in this District, including at least those identified in paragraph 42 above, are places of business of Intel. Intel conducts business from its location(s) in this District, including at least those identified in paragraph 42 above, including but not limited to, making, using, selling, offering for sale, and/or supporting infringing products and services.

d. Intel's location(s) in this District, including at least those identified in paragraph 42 above, are physical, geographical location(s) in this District from which Intel carries out its business.

e. Intel employees work at Intel's location(s), including at least those identified in paragraph 42 above. Upon information and belief, these Intel employees are regularly and physically present at Intel's location(s), including at least those identified in paragraph 42 above, during business hours and they are conducting Intel's business while working there.

FACTUAL ALLEGATIONS

I. PATENTS-IN-SUIT

44. U.S. Patent No. 8,683,296 ("the '8-296 Patent") is entitled "Accelerated Erasure Coding System and Method." The '8-296 Patent duly and legally issued on March 25, 2014, from U.S. Patent Application No. 13/341,833, filed on December 30, 2011. StreamScale is the current owner of all rights, title, and interest in and to the '8-296 Patent. A true and correct copy of the '8-296 Patent is attached hereto as Exhibit A and is incorporated by reference herein.

45. U.S. Patent No. 9,160,374 ("the '374 Patent") is entitled "Accelerated Erasure Coding System and Method." The '374 Patent duly and legally issued on October 13, 2015, from U.S. Patent Application No. 14/223,740, filed on March 24, 2014. The '374 Patent is a continuation of U.S. Patent Application No. 13/341,833, filed on December 30, 2011, now U.S. Patent No. 8,683,296. The '374 Patent is entitled to the benefit of the December 30, 2011 filing date of application No. 13/341,833. StreamScale is the current owner of all rights, title, and interest in and to the '374 Patent. A true and correct copy of the '374 Patent is attached hereto as Exhibit B and is incorporated by reference herein.

46. U.S. Patent No. 9,385,759 ("the '759 Patent") is entitled "Accelerated Erasure Coding System and Method." The '759 Patent duly and legally issued on July 5, 2016, from U.S. Patent Application No. 14/852,438, filed on September 11, 2015. The '759 Patent is a continuation of U.S. Patent Application No. 14/223,740, filed on March 24, 2014, now U.S. Patent

No. 9,160,374. U.S. Patent No. 9,160,374 is a continuation of U.S. Patent Application No. 13/341,833, filed on December 30, 2011, now U.S. Patent No. 8,683,296. The '759 Patent is entitled to the benefit of the December 30, 2011 filing date of application No. 13/341,833. StreamScale is the current owner of all rights, title, and interest in and to the '759 Patent. A true and correct copy of the '759 Patent is attached hereto as Exhibit C and is incorporated by reference herein.

47. U.S. Patent No. 10,003,358 (“the '358 Patent”) is entitled “Accelerated Erasure Coding System and Method.” The '358 Patent duly and legally issued on June 19, 2018, from U.S. Patent Application No. 15/201,196, filed on July 1, 2016. The '358 Patent is a continuation of U.S. Patent Application No. 14/852,438, filed on September 11, 2015, now U.S. Patent No. 9,385,759. U.S. Patent No. 9,385,759 is a continuation of U.S. Patent Application No. 14/223,740, filed on March 24, 2014, now U.S. Patent No. 9,160,374. U.S. Patent No. 9,160,374 is a continuation of U.S. Patent Application No. 13/341,833, filed on December 30, 2011, now U.S. Patent No. 8,683,296. The '358 Patent is entitled to the benefit of the December 30, 2011 filing date of application No. 13/341,833. StreamScale is the current owner of all rights, title, and interest in and to the '358 Patent. A true and correct copy of the '358 Patent is attached hereto as Exhibit D and is incorporated by reference herein. On or about February 23, 2021, StreamScale filed a Petition for Correction of Inventorship Under 37 C.F.R. § 1.324, including associated documentation and fees, with the United States Patent and Trademark Office requesting the correction of inventorship of the '358 Patent to include inventor Sarah Mann, who was not named as an inventor through error. True and correct copies of that Petition and associated documentation are attached as Exhibit E, and that material is incorporated by reference herein.

48. U.S. Patent No. 10,291,259 (“the ’259 Patent”) is entitled “Accelerated Erasure Coding System and Method.” The ’259 Patent duly and legally issued on May 14, 2019, from U.S. Patent Application No. 15/976,175, filed on May 10, 2018. The ’259 Patent is a continuation of U.S. Patent Application No. 15/201,196, filed on July 1, 2016, now U.S. Patent No. 10,003,358. U.S. Patent No. 10,003,358 is a continuation of U.S. Patent Application No. 14/852,438, filed on September 11, 2015, now U.S. Patent No. 9,385,759. U.S. Patent No. 9,385,759 is a continuation of U.S. Patent Application No. 14/223,740, filed on March 24, 2014, now U.S. Patent No. 9,160,374. U.S. Patent No. 9,160,374 is a continuation of U.S. Patent Application No. 13/341,833, filed on December 30, 2011, now U.S. Patent No. 8,683,296. The ’259 Patent is entitled to the benefit of the December 30, 2011 filing date of application No. 13/341,833. StreamScale is the current owner of all rights, title, and interest in and to the ’259 Patent. A true and correct copy of the ’259 Patent is attached hereto as Exhibit F and is incorporated by reference herein. On or about February 23, 2021, StreamScale filed a Petition for Correction of Inventorship Under 37 C.F.R. § 1.324, including associated documentation and fees, with the United States Patent and Trademark Office requesting the correction of inventorship of the ’259 Patent to include inventor Sarah Mann, who was not named as an inventor through error. True and correct copies of that Petition and associated documentation are attached as Exhibit G, and that material is incorporated by reference herein.

49. U.S. Patent No. 10,666,296 (“the ’10-296 Patent”) is entitled “Accelerated Erasure Coding System and Method.” The ’10-296 Patent duly and legally issued on May 26, 2020, from U.S. Patent Application No. 16/358,602, filed on March 19, 2019. The ’10-296 Patent is a continuation of U.S. Patent Application No. 15/976,175, filed on May 10, 2018, now U.S. Patent No. 10,291,259. U.S. Patent No. 10,291,259 is a continuation of U.S. Patent Application

No. 15/201,196, filed on July 1, 2016, now U.S. Patent No. 10,003,358. U.S. Patent No. 10,003,358 is a continuation of U.S. Patent Application No. 14/852,438, filed on September 11, 2015, now U.S. Patent No. 9,385,759. U.S. Patent No. 9,385,759 is a continuation of U.S. Patent Application No. 14/223,740, filed on March 24, 2014, now U.S. Patent No. 9,160,374. U.S. Patent No. 9,160,374 is a continuation of U.S. Patent Application No. 13/341,833, filed on December 30, 2011, now U.S. Patent No. 8,683,296. The '10-296 Patent is entitled to the benefit of the December 30, 2011 filing date of application No. 13/341,833. StreamScale is the current owner of all rights, title, and interest in and to the '10-296 Patent. A true and correct copy of the '10-296 Patent is attached hereto as Exhibit H and is incorporated by reference herein.

50. Collectively, the '8-296 Patent, the '374 Patent, the '759 Patent, the '358 Patent, the '259 Patent, and the '10-296 Patent are referred to herein as the "Patents-in-Suit."

II. ACCELERATED ERASURE CODING INFRINGEMENT

51. As further discussed below, Cloudera, ADP, Experian, and Wargaming (the "EC System Defendants") directly and/or indirectly infringed—and continue to directly and/or indirectly infringe—each of the Patents-in-Suit by engaging in acts constituting infringement under 35 U.S.C. § 271(a) and (b), including without limitation by one or more of making, using, selling, and/or offering to sell, in this District and elsewhere in the United States, and/or importing into this District and elsewhere in the United States, systems that incorporate Cloudera Erasure Coding Components. Cloudera Erasure Coding Components include Cloudera Distribution Including Apache Hadoop ("Cloudera CDH"), which may include any related components, and any Cloudera product or service that is substantially or reasonably similar, including but not limited to Cloudera Enterprise. The infringing systems that Cloudera runs that use the Cloudera Erasure Coding Components are the "Cloudera Infringing Products and Services."

52. Systems built by Cloudera, ADP, Experian, and Wargaming with Cloudera CDH or substantially similar technology include accelerated erasure coding (“EC”) technology are the “EC Systems.” These Defendants are “EC System Defendants.”

53. Under typical configurations, the EC Systems that use the patented technology reduce storage cost by at least about 50% compared with triple replication. Upon information and belief, Cloudera and its collaborators recognized that accelerated erasure coding can greatly reduce storage overhead without sacrificing data reliability, which makes erasure coding a quite attractive alternative for big data storage, particularly for cold data.

54. EC technology is packaged and shipped with Cloudera CDH. Additionally, this EC technology is enabled by default in Cloudera CDH.

55. ADP has directly infringed, and continues to directly infringe, each of the Patents-in-Suit by engaging in acts constituting infringement under 35 U.S.C. § 271(a), including without limitation by one or more of making, using, selling and/or offering to sell, in this District and elsewhere in the United States, and/or importing into this District and elsewhere in the United States, at least ADP’s products and services that use and/or incorporate the Cloudera Erasure Coding Components, including but not limited to DataCloud, and any ADP product or service that is substantially or reasonably similar (the “ADP Infringing Products and Services”).

56. As its name implies, data is core to ADP’s business. Upon information and belief, ADP, a provider of human capital management solutions, is responsible for getting one in six Americans paid today, which puts tremendous data in ADP’s hands.

57. Upon information and belief, ADP is now putting that data to use and generating a new revenue stream. For example, upon information and belief, ADP has built at least a product called DataCloud, which employs Cloudera Erasure Coding Components, that aggregates

information across ADP's 600,000 clients and generates insights to help clients prevent employee churn, ensure salary equality, and maximize human resources.

58. Upon information and belief, ADP was able to use DataCloud to identify the top one percent of at-risk employees in a pilot account, and learned that within that group, turnover was actually 50 percent. When removing that top one percent from the overall analysis, average turnover dropped to nine percent. DataCloud helped the client focus on a small population of at-risk employees where they could make a meaningful impact that would drastically improve the company's overall churn; without this insight, they would have spread retention efforts across the employee base, requiring more time and resources with a less targeted approach and having a lower impact overall.

59. Reducing employee churn has far-reaching business impacts. The cost of losing one employee is more than a simple hiring replacement. Recruiting and interviewing for that person's replacement is costly. Productivity is lost while the new hire gets up to speed. Risk of others on the team leaving increases when they're forced to pick up the slack. It's a ripple effect.

60. The value DataCloud offers is evidenced by the massive growth ADP has seen throughout its client base, driving greater success for ADP via this new revenue channel.

61. Upon information and belief, DataCloud stemmed from a strategic shift at ADP to move from primarily processing transactions to also providing insights based on its greatest asset: data. Upon considering building this product, ADP reached out to clients to gauge their interest in gaining insights based on aggregated and anonymized benchmarks developed from the data spanning ADP's customer base. But making the vision a reality presented a technological challenge. Upon information and belief, ADP's data was spread across data centers and

applications. It needed to be brought together for processing, exploration, and analysis. It wouldn't be feasible using traditional relational database technology.

62. ADP built DataCloud to allow for the storage and processing of large amounts of data in new ways. According to Jim Haas, Principal Architect of DataCloud, advanced data storage and prioritization technologies let companies "maximize CPU time and memory used," which for HR leaders means "getting the big tasks done faster."² KPMG reports that 42% of organizations will replace their existing HR software with a cloud-based solution, with most citing better functionality and higher business value as the motivation.³ But the sheer amount of employee data, devices, access permissions, and historical data needed to effectively track current conditions and develop long-term policies can easily overwhelm standard infrastructure.

63. Upon information and belief, DataCloud employs Cloudera Enterprise, comprising a 200-terabyte (TB) lab and two 400-TB production data centers, each with replication for disaster recovery.

64. Upon information and belief, ten data domains feed DataCloud a billion records every quarter, including: (1) 600,000-plus client databases capturing information on 29 million people; (2) mainframe-based data from the 30 to 35 million pay cycles ADP executes annually, including compensation, time card punches, bonuses, overtime, and salary increases; (3) Oracle-based data from the 15 million HR functions managed by ADP annually, such as benefit deductions and elections, performance scores, and recruiting processes; (4) data from 15 other ADP

² Doug Bonderud, *HR Cloud Solutions: A Foundation for Better Decision Making*, ADP, <https://www.adp.com/spark/articles/2018/01/hr-cloud-solutions-a-foundation-for-better-decision-making.aspx> (last visited Jan. 19, 2021).

³ See, e.g., 2016 HR Transformation Survey: Summary Report, KPMG, <https://assets.kpmg/content/dam/kpmg/in/pdf/2016/11/HR-Transformation-Survey-Summaryreport.pdf> (last visited Jan. 19, 2021).

departments—such as Marketing, Sales, Implementations, and Service—who leverage the platform as their enterprise data hub (EDH) so they may build their own data products; and (5) client data sets such as point-of-sale transactions and revenues.

65. DataCloud conforms job title and role categorizations across 600,000 companies into a comparable standard from which 500 billion aggregates are created. Those aggregates are used to build the benchmarks that are delivered to clients. Upon information and belief, Jim Haas, Principal Architect at ADP has explained, “the data is drawing everybody together Sometimes I call it ‘the little cluster that can’ because it’s just amazing what goes on in there in a day.”

66. Upon information and belief, the ADP Infringing Products and Services are configured to support accelerated erasure coding.

67. Experian has directly infringed, and continues to directly infringe, each of the Patents-in-Suit by engaging in acts constituting infringement under 35 U.S.C. §§ 271(a), including without limitation by one or more of making, using, selling and/or offering to sell, in this District and elsewhere in the United States, and/or importing into this District and elsewhere in the United States, at least Experian’s products and services that use and/or incorporate the Cloudera Infringing Products and Services, including but not limited to Experian Analytical Sandbox and Velcro, and any Experian product or service that is substantially or reasonably similar (the “Experian Infringing Products and Services”).

68. Upon information and belief, Experian integrated Cloudera Enterprise onto its cloud environment for its Credit Information Services, Decision Analytics, and Business Information Services business lines. Upon information and belief, Experian employs Cloudera Erasure Coding Components in Experian’s Ascend Technology Platform and Analytical Sandbox.

69. Experian is doing business in the United States and more particularly in this District, including at least through Experian's corporate subsidiaries CSIdentity Corporation and Experian Information Solutions, Inc., by making, using, selling, importing, and/or offering for sale the product and services that infringe one or more of the patent claims involved in this action.

70. Upon information and belief, with 15,000+ employees and annual revenues exceeding \$4 billion (USD), Experian is a global leader in credit reporting and marketing services that is comprised of four main business units: Credit Information Services, Decision Analytics, Business Information Services, and Marketing Services.

71. Experian Marketing Services ("EMS"), for example, helps marketers connect with customers through relevant communications across a variety of channels, driven by advanced analytics on an extensive database of geographic, demographic, and lifestyle data.

72. EMS has built its business on the effective collection, analysis, and use of data. Upon information and belief, as EMS's former VP of product strategy Jeff Hassemer once explained, "Experian has handled large amounts of data for a very long time: who consumers are, how they're connected, how they interact. We've done this over billions and quadrillions of records over time. But with the proliferation of channels and information that are now flowing into client organizations—social media likes, web interactions, email responses—that data has gotten so large that it's maxed the capacity of older systems. We needed to leap forward in our processing ability. We wanted to process data orders of magnitude faster so we could react to tomorrow's consumer."

73. Today's consumers leave a digital trail of behaviors and preferences for marketers to leverage so they can enhance the customer experience, and upon information and belief,

Experian's clients started asking for more frequent updates on consumers' latest purchasing behaviors, online browsing patterns, and social media activity so they can respond in real time.

74. Upon information and belief, Experian recognized that the data exhaust from these digital channels is massive and requires a technological infrastructure that can accommodate rapid processing, large-scale storage, and flexible analysis of multi-structured data. Experian's mainframes were hitting the tipping point in terms of performance, flexibility, and scalability. Given the need for immediacy of information and customization of data in real time for clients, EMS set an internal goal to process more than 100 million records of data per hour (28,000 records per second).

75. Upon information and belief, instead of trying to fit a square peg in a round hole, Experian went out and decided to build an architecture that could handle the new volumes of data that it manages and built a system that employs Cloudera CDH.

76. Upon information and belief, the Experian Infringing Products and Services are configured to support accelerated erasure coding.

77. Wargaming has directly infringed, and continues to directly infringe, each of the Patents-in-Suit by engaging in acts constituting infringement under 35 U.S.C. § 271(a), including without limitation by one or more of making, using, selling and/or offering to sell, in this District and elsewhere in the United States, and/or importing into this District and elsewhere in the United States, at least Wargaming's products and services that use and/or incorporate Cloudera Erasure Coding Components, including but not limited to Wargaming's Player Relationship Management Platform ("PRMP") in support of Wargaming's online games and massively multiplayer online ("MMO") games, and any Wargaming product or service that is substantially or reasonably similar (the "Wargaming Infringing Products and Services").

78. Wargaming provides strategic intelligence analytics services and coordinates the data services architecture for Wargaming MMO games. Wargaming is a global services hub for games developed, at least in part, by Wargaming Group Limited and accessible via the portal at www.wargaming.net. Wargaming provides data-driven insights, analysis, and reporting of wargaming.net projects, strategic planning, and global game design services through business analytics, production, central technology, and regional administrative departments.

79. Furthermore, Wargaming conducts general research on topics such as the gaming industry, player behavior, and game defects.

80. Wargaming serves more than 150 million registered players in its MMO games. Those millions of players generate massive amounts of data. Wargaming processes over 3 TB of data daily. Upon information and belief, it does this using systems built with Cloudera Erasure Coding Components.

81. Wargaming employees administrate and optimize a series of development and production clusters that employ Cloudera Erasure Coding Components.

82. Upon information and belief, the Wargaming Infringing Products and Services are configured to support accelerated erasure coding.

III. WIDESPREAD KNOWLEDGE OF STREAMSCALE'S PATENTS-IN-SUIT

83. The United States Patent and Trademark Office published the patent application that ultimately led to the '8-296 Patent on July 4, 2013. The very next day, July 5, 2013, StreamScale sent a letter to USENIX, a computing systems association, notifying USENIX of StreamScale's pending patent applications and providing USENIX with advance notice of StreamScale's intent to issue a press release that StreamScale's then-patent-pending technology.⁴

⁴ Exhibit I, Letter from Michael S. Adler, Counsel for StreamScale, to USENIX (July 5, 2013).

Upon information and belief, others in the industry, including but not limited to Intel, learned of StreamScale and its patent applications as a consequence of the letter StreamScale wrote to USENIX.

84. On July 10, 2013, while StreamScale awaited a response from USENIX, Intel publicly announced its excitement to support development of erasure code solutions. Intel explained that erasure codes reduce the size of data on disk by up to half versus traditional replication, decreases costs by more than 50%, and reduces both hardware requirement costs and power and cooling costs.⁵ Intel explained that erasure code was a long overdue technology and Intel was excited to support, promote, and use it in cloud environments.⁶

85. On July 23, 2013, StreamScale issued a press release noting that its technology is protected by then-pending patent applications and was not “open source.”⁷

86. Having received the July 5, 2013 letter that StreamScale sent to USENIX, and following consultation with its attorneys, USENIX chose to comply with StreamScale’s request to remove certain papers and materials from its web site.

87. On or about August 3, 2013, individuals began posting missives online regarding StreamScale and its patent portfolio. Upon information and belief, H. Peter Anvin, an Intel employee was aware of at least some of these online postings. Indeed, upon information and belief,

⁵ Exhibit J, Joe Arnold, Save Space: The Final Frontier—Erasure Codes with OpenStack Swift (July 10, 2013), *previously available at* <https://swiftstack.com/blog/2013/07/10/erasure-codes-with-openstack-swift/>.

⁶ *Id.*

⁷ Exhibit K, StreamScale Provides Notice of Ownership of Fastest Erasure Code Technology Disclosed at Fast ’13 (July 23, 2013).

at least Mr. Anvin commented on at least some of these online postings, including but not limited to on or about August 9, 2013.

88. Upon information and belief, in early March 2014, Intel employees again learned about StreamScale, its patented and patent-pending technology, and its relationship to ISA-L. On March 10, 2014, upon information and belief, one or more Intel employees reviewed and collected a significant quantity of information about StreamScale, its attorneys, and its patent applications. Upon information and belief, one or more Intel employees visited a number of specific pages on StreamScale's website, including (i) those detailing StreamScale's then-pending-patent applications, (ii) those summarizing StreamScale's company history and technology, (iii) those making recent new and press releases available to the public, (iv) those identifying StreamScale's employees and attorneys, and (v) those hosting academic papers authored by StreamScale's employees. Furthermore, upon information and belief, one or more Intel employees accessed and downloaded electronic copies of one or more of StreamScale's patents or patent applications, at least from StreamScale's website.

89. Indeed, upon information and belief, Intel was contacted in February or March 2014 and knew about potential issues involving StreamScale, StreamScale's patent-pending technology, and ISA-L. In August and September 2014, outside counsel for Intel corresponded with then-litigation counsel for StreamScale regarding a third party subpoena StreamScale issued to Intel involving StreamScale's intellectual property rights. Thus, upon information and belief, by mid-to-late September 2014, Intel had knowledge of StreamScale, StreamScale's issued and pending patents and intellectual property rights, and their relevance to ISA-L.

90. Separately, on or about March 5, 2015, Tushar Gohad, an Intel employee, indicated that Jerasure and GF-complete were strategically important. Specifically, Mr. Gohad requested that Jerasure and GF-complete be backported to an earlier version of software. By that time, one of the authors of GF-Complete had publicly stated that StreamScale asserts that the use of GF-Complete (particularly as part of Jerasure 2.0 or later) or any similar software, method or code for erasure coding infringes StreamScale's issued United States Patent No. 8.683,296.

91. On or about April 29, 2015, counsel for StreamScale wrote on an online technical board and asked that the Jerasure 2.0 and GF-Complete libraries that had been republished be removed.⁸ The next day, April 30, 2015, upon information and belief, StreamScale's post and a Techdirt article regarding StreamScale's patent rights were brought to the attention of Paul Luse, another Intel employee, who responded "we are all well aware of the info you passed on :)" Upon information and belief, Mr. Luse then encouraged others to ignore StreamScale, indicating that was always the best option.

92. Since at least March 5, 2021, when Intel was served through its registered agent with the Original Complaint for Patent Infringement in this action, Return of Service, *StreamScale, Inc. v. Cloudera, Inc.*, No. 6:21-cv-00198-ADA (W.D. Tex. Mar. 10, 2021), ECF No. 14, Intel has had express knowledge of each of the Patents-in-Suit and its infringement thereof. Intel continues to actively induce infringement of the StreamScale Patents-in-Suit.

93. On July 7, 2021, StreamScale provided Intel express notice of each of the Patents-in-Suit, its infringement thereof, and the role of ISA-L in its infringement thereof. Exhibit M. Intel continues to actively induce infringement of the StreamScale Patents-in-Suit.

⁸ Exhibit L, Michael A. O'Shea, counsel for StreamScale, post to Ubuntu entitled "StreamScale" (Apr. 29, 2015), available at <https://lists.ubuntu.com/archives/technical-board/2015-April/002100.html> (last visited May 24, 2021).

IV. INTEL'S INFRINGEMENT

94. Upon information and belief, Intel is a Fortune 50 company, with revenues exceeding \$70 billion annually.

95. Intel has a long history with United States patent litigation. Upon information and belief, it employs several attorneys and counsel to manage its offensive and defensive patent litigation docket. In addition, upon information and belief, Intel employs several attorneys to evaluate, manage, and track patent assertions in its industry.

96. In addition, upon information and belief, Intel is a member or client of RPX Corporation ("RPX").

97. RPX offers patent risk management services, including defensive patent buying, acquisition syndication, patent intelligence, and advisory services to its members and clients.

98. Also, upon information and belief, Intel is a member or client of Allied Security Trust ("AST").

99. AST offers patent risk mitigation services to some of the world's biggest technology companies and was created to combat unwanted patent assertions and litigation.

100. Intel has a publicly-known corporate policy forbidding its employees from reading patents held by outside companies or individuals. "Intel's own engineers concede that they avoid reviewing other, non-Intel patents so as to avoid willfully infringing them." *Intel Corp. v. Future Link Sys., LLC*, 268 F. Supp. 3d 605, 623 (D. Del. 2017). Upon information and belief, Intel's policy is designed to avoid possible triple damages for willful infringement.

101. In fact, upon information and belief, Intel has reprimanded its employees for inquiring about others' intellectual property rights, including StreamScale's patents. In early 2014, upon information and belief, Intel reprimanded one of its cloud software architects specifically for suggesting that there was a potential issue with ISA-L in connection with StreamScale.

102. Upon information and belief, Intel has rendered itself willfully blind to StreamScale's Patents-in-Suit and the intellectual property rights of others.

103. As further discussed below, Intel has indirectly infringed, and continues to indirectly infringe, each of the Patents-in-Suit by engaging in acts constituting infringement under 35 U.S.C. § 271(b), including without limitation by actively inducing infringement by the EC System Defendants through the deployment and/or use of Intel's Intelligent Storage Acceleration Library ("ISA-L").

104. Upon information and belief, and as explained above, Intel has been aware of the existence of at least one of the Patents-in-Suit beginning at least in 2014. Intel also obtained actual knowledge of its infringement of the Patents-in-Suit when StreamScale served Intel with its Original Complaint in this action on March 5, 2021. Further, Intel received actual knowledge of its infringement of the Patents-in-Suit when StreamScale sent Intel a cease and desist letter on July 7, 2021, identifying the Patents-in-Suit and Intel's infringement thereof. Exhibit M. All of the Patents-in-Suit are continuations of the application that ultimately issued as the '8-296 Patent.

105. ISA-L comprises a collection of optimized low-level functions used for storage applications.

106. ISA-L is optimized for Intel architecture Intel® 64.

107. ISA-L is packaged and shipped with Cloudera CDH.

108. Intel collaborated with Cloudera to apply erasure coding, including on changes made to the NameNode, DataNode, and the client read and write paths, as well as optimizations using Intel ISA-L to accelerate the encoding and decoding calculations.

109. ISA-L is enabled by default in Cloudera CDH.

110. Intel is doing business in the United States and more particularly in this District by actively inducing infringement by at least Cloudera, ADP, Experian, and Wargaming of StreamScale's Patents-in-Suit through the deployment and support of ISA-L.

COUNT 1—INFRINGEMENT OF THE '8-296 PATENT

111. StreamScale incorporates by reference the allegations set forth in Paragraphs 1–110 of this Complaint as though fully set forth herein.

I. DIRECT INFRINGEMENT

112. In violation of 35 U.S.C. § 271(a), Cloudera, ADP, Experian, and Wargaming are and have been directly infringing one or more of the '8-296 Patent's claims, including at least Claim 1, by making, using, selling, and/or offering for sale in the United States, and/or importing into the United States, without authority, erasure code products and services, including but not limited to those utilizing ISA-L, including without limitation the Cloudera Infringing Products and Services, the ADP Infringing Products and Services, the Experian Infringing Products and Services, and the Wargaming Infringing Products and Services, as described above.

113. The EC System Defendants are infringing claims of the '8-296 Patent, including at least Claim 1, literally and/or pursuant to the doctrine of equivalents.

114. Claim 1 of the '8-296 Patent is directed to a system for accelerated error-correcting code (ECC) processing comprising: a processing core for executing computer instructions and accessing data from a main memory; and a non-volatile storage medium for storing the computer instructions, wherein the processing core, the non-volatile storage medium, and the computer instructions are configured to implement an erasure coding system comprising: a data matrix for holding original data in the main memory; a check matrix for holding check data in the main memory; an encoding matrix for holding first factors in the main memory, the first factors being for encoding the original data into the check data; and a thread for executing on the processing

core and comprising: a parallel multiplier for concurrently multiplying multiple data entries of a matrix by a single factor; and a first sequencer for ordering operations through the data matrix and the encoding matrix using the parallel multiplier to generate the check data.

A. CLOUDERA'S DIRECT INFRINGEMENT

115. As to Cloudera, at least the Cloudera Infringing Products and Services, as defined above, comprise hardware and software components that together practice every element of one or more claims of the '8-296 Patent, including at least Claim 1. The Cloudera Infringing Products and Services are systems capable of performing accelerated ECC. They comprise a processing core, including, for example, one or more Intel, AMD, ARM, and/or PPC64 processing cores. The Cloudera Infringing Products and Services include non-volatile storage (memory) and computer instructions to implement accelerated ECC. The accelerated ECC system of the Cloudera Infringing Products and Services includes a data matrix for holding original data, a check matrix for holding check data, and an encoding matrix for holding first factors, all in memory. The first factors of the encoding matrix are used in the Cloudera Infringing Products and Services to encode the original data into check data. The Cloudera Infringing Products and Services include a thread for executing on the processing core that includes a parallel lookup multiplier and a sequencer for ordering operations through the data matrix and encoding matrix to generate the check data.

B. ADP'S DIRECT INFRINGEMENT

116. As to ADP, at least the ADP Infringing Products and Services, as defined above, comprise hardware and software components that together practice every element of one or more claims of the '8-296 Patent, including at least Claim 1. The ADP Infringing Products and Services are systems capable of performing accelerated ECC. They comprise a processing core, including, for example, one or more Intel, AMD, ARM, and/or PPC64 processing cores. The ADP Infringing Products and Services include non-volatile storage (memory) and computer instructions to

implement accelerated ECC. The accelerated ECC system of the ADP Infringing Products and Services includes a data matrix for holding original data, a check matrix for holding check data, and an encoding matrix for holding first factors, all in memory. The first factors of the encoding matrix are used in the ADP Infringing Products and Services to encode the original data into check data. The ADP Infringing Products and Services include a thread for executing on the processing core that includes a parallel lookup multiplier and a sequencer for ordering operations through the data matrix and encoding matrix to generate the check data.

C. EXPERIAN'S DIRECT INFRINGEMENT

117. As to Experian, at least the Experian Infringing Products and Services, as defined above, comprise hardware and software components that together practice every element of one or more claims of the '8-296 Patent, including at least Claim 1. The Experian Infringing Products and Services are systems capable of performing accelerated ECC. They comprise a processing core, including, for example, one or more Intel, AMD, ARM, and/or PPC64 processing cores. The Experian Infringing Products and Services include non-volatile storage (memory) and computer instructions to implement accelerated ECC. The accelerated ECC system of the Experian Infringing Products and Services includes a data matrix for holding original data, a check matrix for holding check data, and an encoding matrix for holding first factors, all in memory. The first factors of the encoding matrix are used in the Experian Infringing Products and Services to encode the original data into check data. The Experian Infringing Products and Services include a thread for executing on the processing core that includes a parallel lookup multiplier and a sequencer for ordering operations through the data matrix and encoding matrix to generate the check data.

D. WARGAMING'S DIRECT INFRINGEMENT

118. As to Wargaming, at least the Wargaming Infringing Products and Services, as defined above, comprise hardware and software components that together practice every element

of one or more claims of the '8-296 Patent, including at least Claim 1. The Wargaming Infringing Products and Services are systems capable of performing accelerated ECC. They comprise a processing core, including, for example, one or more Intel, AMD, ARM, and/or PPC64 processing cores. The Wargaming Infringing Products and Services include non-volatile storage (memory) and computer instructions to implement accelerated ECC. The accelerated ECC system of the Wargaming Infringing Products and Services includes a data matrix for holding original data, a check matrix for holding check data, and an encoding matrix for holding first factors, all in memory. The first factors of the encoding matrix are used in the Wargaming Infringing Products and Services to encode the original data into check data. The Wargaming Infringing Products and Services include a thread for executing on the processing core that includes a parallel lookup multiplier and a sequencer for ordering operations through the data matrix and encoding matrix to generate the check data.

II. INDIRECT INFRINGEMENT

119. In violation of 35 U.S.C. §§ 271(b), Intel is and has been infringing one or more of the '8-296 Patent's claims, including at least Claim 1, indirectly by inducing the infringement of at least Claim 1 of the '8-296 Patent by third parties, including for example Cloudera, ADP, Experian, and Wargaming, in this District and elsewhere in the United States. Direct infringement is the result of activities performed by users of systems that incorporate, among other features, ISA-L, including for example Cloudera, ADP, Experian, and Wargaming, in accordance with at least Claim 1 of the '8-296 Patent.

120. Intel's affirmative acts of selling and/or distributing ISA-L (or portions thereof), causing ISA-L (or portions thereof) to be manufactured and distributed, providing instructive materials and information concerning operation and use of ISA-L (or portions thereof), and providing maintenance/service for such products or services, induced Cloudera, ADP, Experian,

and Wargaming to infringe at least Claim 1 of the '8-296 Patent. For example, Intel induced Cloudera, ADP, Experian, and Wargaming to infringe at least Claim 1 of the '8-296 Patent through the implementation of ISA-L in the Cloudera, ADP, Experian, and Wargaming Infringing Products and Services. By and through these acts, Intel knowingly and specifically intended the users of ISA-L (or portions thereof) to infringe at least Claim 1 of the '8-296 Patent. Intel (1) knew or should have known of the '8-296 Patent since at least 2014, (2) performed and continues to perform affirmative acts that constitute induced infringement, and (3) knew or should have known that those acts would induce actual infringement of one or more of the '8-296 Patent's claims by users of ISA-L.

121. Intel actively markets and instructs the EC System Defendants to create EC Systems using ISA-L.

122. For example, upon information and belief, Intel (i) maintains a website to promote ISA-L,⁹ including to the EC System Defendants, (ii) produces videos regarding ISA-L and its use that are available to the EC System Defendants on the Intel website,¹⁰ (iii) describes case studies on big data optimization using ISA-L that are available to the EC System Defendants on the Intel website, (iv) hosts articles, blog posts, and webinars regarding the use of ISA-L that are available to the EC System Defendants on the Intel website, and (v) publishes and makes available an API Reference Manual for ISA-L¹¹ that is available to the EC System Defendants, which it updates

⁹ *E.g.*, Intel, Intel® Intelligent Storage Acceleration Library, *available at* <https://software.intel.com/content/www/us/en/develop/tools/isa-l.html> (last visited May 24, 2021).

¹⁰ *See, e.g.*, Intel, Erasure Code and Intel® Intelligent Storage Acceleration Library (Intel® ISA-L, *available at* <https://www.intel.com/content/www/us/en/products/docs/storage/erasure-code-isa-l-solution-video.html> (last visited May 24, 2021).

¹¹ *See, e.g.*, Intel, Intel® Intelligent Storage Acceleration Library (Intel® ISA-L) Open Source Version, API Reference Manual (ver. 2.8, Sept. 27, 2013), *available at*

regularly.¹² Upon information and belief, Intel further offers the EC System Defendants technical support for ISA-L and the EC System Defendants' products.

123. Upon information and belief, Intel promotes and encourages the EC System Defendants to use ISA-L in order to drive sales of other Intel products and services to the EC System Defendants.

124. As to Intel, at least ISA-L, as defined above, is designed to be used with other components that, when combined with hardware, practice one or more claims of the '8-296 Patent, including at least Claim 1. EC Systems that employ ISA-L create a data matrix for holding original data, a check matrix for holding check data, and an encoding matrix for holding first factors, all in memory. The first factors of the encoding matrix are used to encode the original data into check data. The systems also include a thread for executing on the processing core that includes a parallel lookup multiplier and a sequencer for ordering operations through the data matrix and encoding matrix to generate the check data.

125. As explained above, Intel had actual knowledge of the '8-296 Patent prior to this lawsuit being filed and had knowledge of the infringing nature of its activities, and the role of ISA-L in that infringement of the '8-296 Patent, yet continues to induce infringement of at least Claim 1 of the '8-296 Patent by Cloudera, ADP, Experian, and Wargaming.

126. Especially in light of its actual knowledge of StreamScale and StreamScale's patent portfolio, Intel subjectively believed there was a high probability that StreamScale's

https://01.org/sites/default/files/documentation/isa-l_open_src_2.8_0.pdf (last visited May 24, 2021).

¹² See, e.g., Intel, Intel® Intelligent Storage Acceleration Library (Intel® ISA-L), API Reference Manual (ver. 2.23.0, June 29, 2018), available at https://01.org/sites/default/files/documentation/isa-l_api_2.23.0.pdf (last visited May 24, 2021).

Patents-in-Suit implicated ISA-L and that EC System Defendants use of ISA-L would infringe StreamScale's Patents-in-Suit, including the '8-296 Patent. To the extent that Intel lacked actual knowledge of the '8-296 Patent or the EC System Defendants' actual infringement of the '8-296 Patent, Intel took deliberate actions to avoid learning of those facts. Indeed, Intel actively encouraged others to ignore StreamScale and its patents and further reprimanded at least one employee for failing to ignore StreamScale and its patents.

127. At a minimum, Intel has had actual notice of the '8-296 Patent since March 5, 2021 and has knowledge of the infringing nature of its activities, yet continues to induce infringement of at least Claim 1 of the '8-296 Patent by Cloudera, ADP, Experian, and Wargaming.

128. Despite knowing of the '8-296 Patent since at least as early as 2014, but in no event later than March 5, 2021, upon information and belief, Intel has never undertaken any serious investigation to form a good faith belief as to non-infringement or invalidity of the '8-296 Patent.

129. Despite knowing of the '8-296 Patent since at least as early as March 5, 2021, Intel has continued to infringe one or more claims of the '8-296 Patent.

130. Despite knowing of the '8-296 Patent since at least July 7, 2021, Intel has continued to infringe one or more claims of the '8-296 Patent.

131. Therefore, upon information and belief, Intel's infringement of at least Claim 1 of the '8-296 Patent has been and continues to be willful, wanton, malicious, bad-faith, deliberate, consciously wrongful, flagrant, or characteristic of a pirate, entitling StreamScale to increased damages pursuant to 35 U.S.C. § 284 and to attorneys' fees and costs incurred in prosecuting this action pursuant to 35 U.S.C. § 285.

III. DAMAGES

132. Defendants' acts of infringement have caused damages to StreamScale, and StreamScale is entitled to recover from Defendants the damages sustained by StreamScale as a result of Defendants' wrongful acts in an amount to be determined at trial.

COUNT 2—INFRINGEMENT OF THE '374 PATENT

133. StreamScale incorporates by reference the allegations set forth in Paragraphs 1–132 of this Complaint as though fully set forth herein.

I. DIRECT INFRINGEMENT

134. In violation of 35 U.S.C. § 271(a), Cloudera, ADP, Experian, and Wargaming are and have been directly infringing one or more of the '374 Patent's claims, including at least Claim 1, by making, using, selling, and/or offering for sale in the United States, and/or importing into the United States, without authority, erasure code products and services, including but not limited to those utilizing ISA-L, including without limitation the Cloudera Infringing Products and Services, the ADP Infringing Products and Services, the Experian Infringing Products and Services, and the Wargaming Infringing Products and Services, as described above.

135. The EC System Defendants are infringing claims of the '374 Patent, including at least Claim 1, literally and/or pursuant to the doctrine of equivalents.

136. Claim 1 of the '374 Patent is directed to a system for accelerated error-correcting code (ECC) processing comprising: a processing core for executing computer instructions and accessing data from a main memory, the processing core comprising at least 16 data registers, each of the data registers comprising at least 16 bytes; and a non-volatile storage medium for storing the computer instructions, wherein the processing core, the non-volatile storage medium, and the computer instructions are configured to implement an erasure coding system comprising: a data matrix for holding original data in the main memory; a check matrix for holding check data in the

main memory; an encoding matrix for holding first factors in the main memory, the first factors being for encoding the original data into the check data; and a thread for executing on the processing core and comprising: a parallel multiplier for concurrently multiplying multiple data entries of a matrix by a single factor; and a first sequencer for ordering operations through the data matrix and the encoding matrix using the parallel multiplier to generate the check data.

A. CLOUDERA'S DIRECT INFRINGEMENT

137. As to Cloudera, at least the Cloudera Infringing Products and Services, as defined above, comprise hardware and software components that together practice every element of one or more claims of the '374 Patent, including at least Claim 1. The Cloudera Infringing Products and Services are systems capable of performing accelerated ECC. They comprise a processing core comprising at least 16 data registers of at least 16 bytes each, including, for example, one or more Intel, AMD, ARM, and/or PPC64 processing cores. The Cloudera Infringing Products and Services include non-volatile storage (memory) and computer instructions to implement accelerated ECC. The accelerated ECC system of the Cloudera Infringing Products and Services includes a data matrix for holding original data, a check matrix for holding check data, and an encoding matrix for holding first factors, all in memory. The first factors of the encoding matrix are used in the Cloudera Infringing Products and Services to encode the original data into check data. The Cloudera Infringing Products and Services include a thread for executing on the processing core that includes a parallel lookup multiplier and a sequencer for ordering operations through the data matrix and encoding matrix to generate the check data.

B. ADP'S DIRECT INFRINGEMENT

138. As to ADP, at least the ADP Infringing Products and Services, as defined above, comprise hardware and software components that together practice every element of one or more claims of the '374 Patent, including at least Claim 1. The ADP Infringing Products and Services

are systems capable of performing accelerated ECC. They comprise a processing core comprising at least 16 data registers of at least 16 bytes each, including, for example, one or more Intel, AMD, ARM, and/or PPC64 processing cores. The ADP Infringing Products and Services include non-volatile storage (memory) and computer instructions to implement accelerated ECC. The accelerated ECC system of the ADP Infringing Products and Services includes a data matrix for holding original data, a check matrix for holding check data, and an encoding matrix for holding first factors, all in memory. The first factors of the encoding matrix are used in the ADP Infringing Products and Services to encode the original data into check data. The ADP Infringing Products and Services include a thread for executing on the processing core that includes a parallel lookup multiplier and a sequencer for ordering operations through the data matrix and encoding matrix to generate the check data.

C. EXPERIAN'S DIRECT INFRINGEMENT

139. As to Experian, at least the Experian Infringing Products and Services, as defined above, comprise hardware and software components that together practice every element of one or more claims of the '374 Patent, including at least Claim 1. The Experian Infringing Products and Services are systems capable of performing accelerated ECC. They comprise a processing core comprising at least 16 data registers of at least 16 bytes each, including, for example, one or more Intel, AMD, ARM, and/or PPC64 processing cores. The Experian Infringing Products and Services include non-volatile storage (memory) and computer instructions to implement accelerated ECC. The accelerated ECC system of the Experian Infringing Products and Services includes a data matrix for holding original data, a check matrix for holding check data, and an encoding matrix for holding first factors, all in memory. The first factors of the encoding matrix are used in the Experian Infringing Products and Services to encode the original data into check data. The Experian Infringing Products and Services include a thread for executing on the

processing core that includes a parallel lookup multiplier and a sequencer for ordering operations through the data matrix and encoding matrix to generate the check data.

D. WARGAMING'S DIRECT INFRINGEMENT

140. As to Wargaming, at least the Wargaming Infringing Products and Services, as defined above, comprise hardware and software components that together practice every element of one or more claims of the '374 Patent, including at least Claim 1. The Wargaming Infringing Products and Services are systems capable of performing accelerated ECC. They comprise a processing core comprising at least 16 data registers of at least 16 bytes each, including, for example, one or more Intel, AMD, ARM, and/or PPC64 processing cores. The Wargaming Infringing Products and Services include non-volatile storage (memory) and computer instructions to implement accelerated ECC. The accelerated ECC system of the Wargaming Infringing Products and Services includes a data matrix for holding original data, a check matrix for holding check data, and an encoding matrix for holding first factors, all in memory. The first factors of the encoding matrix are used in the Wargaming Infringing Products and Services to encode the original data into check data. The Wargaming Infringing Products and Services include a thread for executing on the processing core that includes a parallel lookup multiplier and a sequencer for ordering operations through the data matrix and encoding matrix to generate the check data.

II. INDIRECT INFRINGEMENT

141. In violation of 35 U.S.C. §§ 271(b), Intel is and has been infringing one or more of the '374 Patent's claims, including at least Claim 1, indirectly by inducing the infringement of at least Claim 1 of the '374 Patent by third parties, including for example Cloudera, ADP, Experian, and Wargaming, in this District and elsewhere in the United States. Direct infringement is the result of activities performed by users of systems that incorporate, among other features, ISA-L,

including for example Cloudera, ADP, Experian, and Wargaming, in accordance with at least Claim 1 of the '374 Patent.

142. Intel's affirmative acts of selling and/or distributing ISA-L (or portions thereof), causing ISA-L (or portions thereof) to be manufactured and distributed, providing instructive materials and information concerning operation and use of ISA-L (or portions thereof), and providing maintenance/service for such products or services, induced Cloudera, ADP, Experian, and Wargaming to infringe at least Claim 1 of the '374 Patent. For example, Intel induced Cloudera, ADP, Experian, and Wargaming to infringe at least Claim 1 of the '374 Patent through the implementation of ISA-L in the Cloudera, ADP, Experian, and Wargaming Infringing Products and Services. By and through these acts, Intel knowingly and specifically intended the users of ISA-L (or portions thereof) to infringe at least Claim 1 of the '374 Patent. Intel (1) knew or should have known of the '374 Patent since at least 2015, (2) performed and continues to perform affirmative acts that constitute induced infringement, and (3) knew or should have known that those acts would induce actual infringement of one or more of the '374 Patent's claims by users of ISA-L.

143. For example, upon information and belief, Intel (i) maintains a website to promote ISA-L,¹³ including to the EC System Defendants, (ii) produces videos regarding ISA-L and its use that are available to the EC System Defendants on the Intel website,¹⁴ (iii) describes case studies on big data optimization using ISA-L that are available to the EC System Defendants on the Intel

¹³ *E.g.*, Intel, Intel® Intelligent Storage Acceleration Library, *available at* <https://software.intel.com/content/www/us/en/develop/tools/isa-l.html> (last visited May 24, 2021).

¹⁴ *See, e.g.*, Intel, Erasure Code and Intel® Intelligent Storage Acceleration Library (Intel® ISA-L, *available at* <https://www.intel.com/content/www/us/en/products/docs/storage/erasure-code-isa-l-solution-video.html> (last visited May 24, 2021).

website, (iv) hosts articles, blog posts, and webinars regarding the use of ISA-L that are available to the EC System Defendants on the Intel website, and (v) publishes and makes available an API Reference Manual for ISA-L¹⁵ that is available to the EC System Defendants, which it updates regularly.¹⁶ Upon information and belief, Intel further offers the EC System Defendants technical support for ISA-L and the EC System Defendants' products.

144. Upon information and belief, Intel promotes and encourages the EC System Defendants to use ISA-L in order to drive sales of other Intel products and services to the EC System Defendants.

145. As to Intel, at least ISA-L, as defined above, is designed to be used with other components that, when combined with hardware, practice one or more claims of the '374 Patent, including at least Claim 1. EC Systems that employ ISA-L create a data matrix for holding original data, a check matrix for holding check data, and an encoding matrix for holding first factors, all in memory. The first factors of the encoding matrix are used to encode the original data into check data. The systems also include a thread for executing on the processing core that includes a parallel lookup multiplier and a sequencer for ordering operations through the data matrix and encoding matrix to generate the check data.

146. Especially in light of its actual knowledge of StreamScale and StreamScale's patent portfolio, Intel subjectively believed there was a high probability that StreamScale's

¹⁵ See, e.g., Intel, Intel® Intelligent Storage Acceleration Library (Intel® ISA-L) Open Source Version, API Reference Manual (ver. 2.8, Sept. 27, 2013), available at https://01.org/sites/default/files/documentation/isa-l_open_src_2.8_0.pdf (last visited May 24, 2021).

¹⁶ See, e.g., Intel, Intel® Intelligent Storage Acceleration Library (Intel® ISA-L), API Reference Manual (ver. 2.23.0, June 29, 2018), available at https://01.org/sites/default/files/documentation/isa-l_api_2.23.0.pdf (last visited May 24, 2021).

Patents-in-Suit implicated ISA-L and that EC System Defendants use of ISA-L would infringe StreamScale's Patents-in-Suit, including the '374 Patent. To the extent that Intel lacked actual knowledge of the '374 Patent or the EC System Defendants' actual infringement of the '374 Patent, Intel took deliberate actions to avoid learning of those facts. Indeed, Intel actively encouraged others to ignore StreamScale and its patents and further reprimanded at least one employee for failing to ignore StreamScale and its patents.

147. At a minimum, Intel has had actual notice of the '374 Patent since March 5, 2021 and has knowledge of the infringing nature of its activities, yet continues to induce infringement of at least Claim 1 of the '374 Patent by Cloudera, ADP, Experian, and Wargaming.

148. Despite knowing of the 374 Patent since at least as early as March 5, 2021, upon information and belief, Intel has never undertaken any serious investigation to form a good faith belief as to non-infringement or invalidity of the '374 Patent.

149. Despite knowing of the '374 Patent since at least as early as March 5, 2021, Intel has continued to infringe one or more claims of the '374 Patent.

150. Despite knowing of the '374 Patent since at least July 7, 2021, Intel has continued to infringe one or more claims of the '374 Patent.

151. Therefore, upon information and belief, Intel's infringement of at least Claim 1 of the '374 Patent has been and continues to be willful, wanton, malicious, bad-faith, deliberate, consciously wrongful, flagrant, or characteristic of a pirate, entitling StreamScale to increased damages pursuant to 35 U.S.C. § 284 and to attorneys' fees and costs incurred in prosecuting this action pursuant to 35 U.S.C. § 285.

III. DAMAGES

152. Defendants' acts of infringement have caused damages to StreamScale, and StreamScale is entitled to recover from Defendants the damages sustained by StreamScale as a result of Defendants' wrongful acts in an amount to be determined at trial.

COUNT 3—INFRINGEMENT OF THE '759 PATENT

153. StreamScale incorporates by reference the allegations set forth in Paragraphs 1–152 of this Complaint as though fully set forth herein.

I. DIRECT INFRINGEMENT

154. In violation of 35 U.S.C. § 271(a), Cloudera, ADP, Experian, and Wargaming are and have been directly infringing one or more of the '759 Patent's claims, including at least Claim 1, by making, using, selling, and/or offering for sale in the United States, and/or importing into the United States, without authority, erasure code products and services, including but not limited to those utilizing ISA-L, including without limitation the Cloudera Infringing Products and Services, the ADP Infringing Products and Services, the Experian Infringing Products and Services, and the Wargaming Infringing Products and Services, as described above.

155. The EC System Defendants are infringing claims of the '759 Patent, including at least Claim 1, literally and/or pursuant to the doctrine of equivalents.

156. Claim 1 of the '759 Patent is directed to a system for accelerated error-correcting code (ECC) processing comprising: a processing core for executing computer instructions and accessing data from a main memory, the processing core comprising at least 16 data registers, each of the data registers comprising at least 16 bytes; one or more non-volatile storage media for storing the computer instructions and the data; and an input/output (I/O) controller for controlling data transfers between the main memory and the non-volatile storage media, wherein the processing core, the non-volatile storage media, the I/O controller, and the computer instructions are

configured to implement an erasure coding system comprising: a data matrix for holding original data in the main memory; a check matrix for holding check data in the main memory; an encoding matrix for holding first factors in the main memory, the first factors being for encoding the original data into the check data; and a thread for executing on the processing core and comprising: a parallel multiplier for concurrently multiplying multiple data entries of a matrix by a single factor; and a first sequencer for ordering data accesses through the data matrix and the encoding matrix using the parallel multiplier to generate the check data.

A. CLOUDERA'S DIRECT INFRINGEMENT

157. As to Cloudera, at least the Cloudera Infringing Products and Services, as defined above, comprise hardware and software components that together practice every element of one or more claims of the '759 Patent, including at least Claim 1. The Cloudera Infringing Products and Services are systems capable of performing accelerated ECC. They comprise a processing core comprising at least 16 data registers of at least 16 bytes each, including, for example, one or more Intel, AMD, ARM, and/or PPC64 processing cores. The Cloudera Infringing Products and Services include non-volatile storage (memory) for storing computer instructions and data. The Cloudera Infringing Products and Services further include an input/output (I/O) controller to coordinate communication and data transfers between the main memory and the non-volatile storage media. The processing core, memory, I/O controller, and computer instructions of the Cloudera Infringing Products and Services implement accelerated ECC. The accelerated ECC system of the Cloudera Infringing Products and Services includes a data matrix for holding original data, a check matrix for holding check data, and an encoding matrix for holding first factors, all in memory. The first factors of the encoding matrix are used in the Cloudera Infringing Products and Services to encode the original data into check data. The Cloudera Infringing Products and Services include a thread for executing on the processing core that includes a parallel lookup

multiplier and a sequencer for ordering data accesses through the data matrix and encoding matrix to generate the check data.

B. ADP'S DIRECT INFRINGEMENT

158. As to ADP, at least the ADP Infringing Products and Services, as defined above, comprise hardware and software components that together practice every element of one or more claims of the '759 Patent, including at least Claim 1. The ADP Infringing Products and Services are systems capable of performing accelerated ECC. They comprise a processing core comprising at least 16 data registers of at least 16 bytes each, including, for example, one or more Intel, AMD, ARM, and/or PPC64 processing cores. The ADP Infringing Products and Services include non-volatile storage (memory) for storing computer instructions and data. The ADP Infringing Products and Services further include an input/output (I/O) controller to coordinate communication and data transfers between the main memory and the non-volatile storage media. The processing core, memory, I/O controller, and computer instructions of the ADP Infringing Products and Services implement accelerated ECC. The accelerated ECC system of the ADP Infringing Products and Services includes a data matrix for holding original data, a check matrix for holding check data, and an encoding matrix for holding first factors, all in memory. The first factors of the encoding matrix are used in the ADP Infringing Products and Services to encode the original data into check data. The ADP Infringing Products and Services include a thread for executing on the processing core that includes a parallel lookup multiplier and a sequencer for ordering data accesses through the data matrix and encoding matrix to generate the check data.

C. EXPERIAN'S DIRECT INFRINGEMENT

159. As to Experian, at least the Experian Infringing Products and Services, as defined above, comprise hardware and software components that together practice every element of one or more claims of the '759 Patent, including at least Claim 1. The Experian Infringing Products

and Services are systems capable of performing accelerated ECC. They comprise a processing core comprising at least 16 data registers of at least 16 bytes each, including, for example, one or more Intel, AMD, ARM, and/or PPC64 processing cores. The Experian Infringing Products and Services include non-volatile storage (memory) for storing computer instructions and data. The Experian Infringing Products and Services further include an input/output (I/O) controller to coordinate communication and data transfers between the main memory and the non-volatile storage media. The processing core, memory, I/O controller, and computer instructions of the Experian Infringing Products and Services implement accelerated ECC. The accelerated ECC system of the Experian Infringing Products and Services includes a data matrix for holding original data, a check matrix for holding check data, and an encoding matrix for holding first factors, all in memory. The first factors of the encoding matrix are used in the Experian Infringing Products and Services to encode the original data into check data. The Experian Infringing Products and Services include a thread for executing on the processing core that includes a parallel lookup multiplier and a sequencer for ordering data accesses through the data matrix and encoding matrix to generate the check data.

D. WARGAMING'S DIRECT INFRINGEMENT

160. As to Wargaming, at least the Wargaming Infringing Products and Services, as defined above, comprise hardware and software components that together practice every element of one or more claims of the '759 Patent, including at least Claim 1. The Wargaming Infringing Products and Services are systems capable of performing accelerated ECC. They comprise a processing core comprising at least 16 data registers of at least 16 bytes each, including, for example, one or more Intel, AMD, ARM, and/or PPC64 processing cores. The Wargaming Infringing Products and Services include non-volatile storage (memory) for storing computer instructions and data. The Wargaming Infringing Products and Services further include an

input/output (I/O) controller to coordinate communication and data transfers between the main memory and the non-volatile storage media. The processing core, memory, I/O controller, and computer instructions of the Wargaming Infringing Products and Services implement accelerated ECC. The accelerated ECC system of the Wargaming Infringing Products and Services includes a data matrix for holding original data, a check matrix for holding check data, and an encoding matrix for holding first factors, all in memory. The first factors of the encoding matrix are used in the Wargaming Infringing Products and Services to encode the original data into check data. The Wargaming Infringing Products and Services include a thread for executing on the processing core that includes a parallel lookup multiplier and a sequencer for ordering data accesses through the data matrix and encoding matrix to generate the check data.

II. INDIRECT INFRINGEMENT

161. In violation of 35 U.S.C. §§ 271(b), Intel is and has been infringing one or more of the '759 Patent's claims, including at least Claim 1, indirectly by inducing the infringement of at least Claim 1 of the '759 Patent by third parties, including for example Cloudera, ADP, Experian, and Wargaming, in this District and elsewhere in the United States. Direct infringement is the result of activities performed by users of systems that incorporate, among other features, ISA-L, including for example Cloudera, ADP, Experian, and Wargaming, in accordance with at least Claim 1 of the '759 Patent.

162. Intel's affirmative acts of selling and/or distributing ISA-L (or portions thereof), causing ISA-L (or portions thereof) to be manufactured and distributed, providing instructive materials and information concerning operation and use of ISA-L (or portions thereof), and providing maintenance/service for such products or services, induced Cloudera, ADP, Experian, and Wargaming to infringe at least Claim 1 of the '759 Patent. For example, Intel induced Cloudera, ADP, Experian, and Wargaming to infringe at least Claim 1 of the '759 Patent through

the implementation of ISA-L in the Cloudera, ADP, Experian, and Wargaming Infringing Products and Services. By and through these acts, Intel knowingly and specifically intended the users of ISA-L (or portions thereof) to infringe at least Claim 1 of the '759 Patent. Intel (1) knew or should have known of the '759 Patent since at least 2016, (2) performed and continues to perform affirmative acts that constitute induced infringement, and (3) knew or should have known that those acts would induce actual infringement of one or more of the '759 Patent's claims by users of ISA-L.

163. For example, upon information and belief, Intel (i) maintains a website to promote ISA-L,¹⁷ including to the EC System Defendants, (ii) produces videos regarding ISA-L and its use that are available to the EC System Defendants on the Intel website,¹⁸ (iii) describes case studies on big data optimization using ISA-L that are available to the EC System Defendants on the Intel website, (iv) hosts articles, blog posts, and webinars regarding the use of ISA-L that are available to the EC System Defendants on the Intel website, and (v) publishes and makes available an API Reference Manual for ISA-L¹⁹ that is available to the EC System Defendants, which it updates

¹⁷ *E.g.*, Intel, Intel® Intelligent Storage Acceleration Library, *available at* <https://software.intel.com/content/www/us/en/develop/tools/isa-l.html> (last visited May 24, 2021).

¹⁸ *See, e.g.*, Intel, Erasure Code and Intel® Intelligent Storage Acceleration Library (Intel® ISA-L), *available at* <https://www.intel.com/content/www/us/en/products/docs/storage/erasure-code-isa-l-solution-video.html> (last visited May 24, 2021).

¹⁹ *See, e.g.*, Intel, Intel® Intelligent Storage Acceleration Library (Intel® ISA-L) Open Source Version, API Reference Manual (ver. 2.8, Sept. 27, 2013), *available at* https://01.org/sites/default/files/documentation/isa-l_open_src_2.8_0.pdf (last visited May 24, 2021).

regularly.²⁰ Upon information and belief, Intel further offers the EC System Defendants technical support for ISA-L and the EC System Defendants' products.

164. Upon information and belief, Intel promotes and encourages the EC System Defendants to use ISA-L in order to drive sales of other Intel products and services to the EC System Defendants.

165. As to Intel, at least ISA-L, as defined above, is designed to be used with other components that, when combined with hardware, practice one or more claims of the '759 Patent, including at least Claim 1. EC Systems that employ ISA-L create a data matrix for holding original data, a check matrix for holding check data, and an encoding matrix for holding first factors, all in memory. The first factors of the encoding matrix are used to encode the original data into check data. The systems also include a thread for executing on the processing core that includes a parallel lookup multiplier and a sequencer for ordering data accesses through the data matrix and encoding matrix to generate the check data.

166. Especially in light of its actual knowledge of StreamScale and StreamScale's patent portfolio, Intel subjectively believed there was a high probability that StreamScale's Patents-in-Suit implicated ISA-L and that EC System Defendants use of ISA-L would infringe StreamScale's Patents-in-Suit, including the '759 Patent. To the extent that Intel lacked actual knowledge of the '759 Patent or the EC System Defendants' actual infringement of the '759 Patent, Intel took deliberate actions to avoid learning of those facts. Indeed, Intel actively encouraged others to ignore StreamScale and its patents and further reprimanded at least one employee for failing to ignore StreamScale and its patents.

²⁰ See, e.g., Intel, Intel® Intelligent Storage Acceleration Library (Intel® ISA-L), API Reference Manual (ver. 2.23.0, June 29, 2018), available at https://01.org/sites/default/files/documentation/isa-l_api_2.23.0.pdf (last visited May 24, 2021).

167. At a minimum, Intel has had actual notice of the '759 Patent since March 5, 2021 and has knowledge of the infringing nature of its activities, yet continues to induce infringement of at least Claim 1 of the '759 Patent by Cloudera, ADP, Experian, and Wargaming.

168. Despite knowing of the '759 Patent since at least as early as March 5, 2021, upon information and belief, Intel has never undertaken any serious investigation to form a good faith belief as to non-infringement or invalidity of the '759 Patent.

169. Despite knowing of the '759 Patent since at least as early as March 5, 2021, Intel has continued to infringe one or more claims of the '759 Patent.

170. Despite knowing of the '759 Patent since at least July 7, 2021, Intel has continued to infringe one or more claims of the '759 Patent.

171. Therefore, upon information and belief, Intel's infringement of at least Claim 1 of the '759 Patent has been and continues to be willful, wanton, malicious, bad-faith, deliberate, consciously wrongful, flagrant, or characteristic of a pirate, entitling StreamScale to increased damages pursuant to 35 U.S.C. § 284 and to attorneys' fees and costs incurred in prosecuting this action pursuant to 35 U.S.C. § 285.

III. DAMAGES

172. Defendants' acts of infringement have caused damages to StreamScale, and StreamScale is entitled to recover from Defendants the damages sustained by StreamScale as a result of Defendants' wrongful acts in an amount to be determined at trial.

COUNT 4—INFRINGEMENT OF THE '358 PATENT

173. StreamScale incorporates by reference the allegations set forth in Paragraphs 1–172 of this Complaint as though fully set forth herein.

I. DIRECT INFRINGEMENT

174. In violation of 35 U.S.C. § 271(a), Cloudera, ADP, Experian, and Wargaming are and have been directly infringing one or more of the '358 Patent's claims, including at least Claim 1, by making, using, selling, and/or offering for sale in the United States, and/or importing into the United States, without authority, erasure code products and services, including but not limited to those utilizing ISA-L, including without limitation the Cloudera Infringing Products and Services, the ADP Infringing Products and Services, the Experian Infringing Products and Services, and the Wargaming Infringing Products and Services, as described above.

175. The EC System Defendants are infringing claims of the '358 Patent, including at least Claim 1, literally and/or pursuant to the doctrine of equivalents.

176. Claim 1 of the '358 Patent is directed to a system adapted to use accelerated error-correcting code (ECC) processing to improve the storage and retrieval of digital data distributed across a plurality of drives, comprising: at least one processor comprising at least one single-instruction-multiple-data (SIMD) central processing unit (CPU) core that executes SIMD instructions and loads original data from a main memory and stores check data to the main memory, the SIMD CPU core comprising at least 16 vector registers, each of the vector registers storing at least 16 bytes; at least one system drive comprising at least one non-volatile storage medium that stores the SIMD instructions; a plurality of data drives each comprising at least one non-volatile storage medium that stores at least one block of the original data, the at least one block comprising at least 512 bytes; more than two check drives each comprising at least one non-volatile storage medium that stores at least one block of the check data; and at least one input/output (I/O) controller that stores the at least one block of the check data from the main memory to the check drives, wherein the processor, the SIMD instructions, the non-volatile storage media, and the I/O controller are configured to implement an erasure coding system comprising: a data matrix

comprising at least one vector and comprising a plurality of rows of at least one block of the original data in the main memory, each of the rows being stored on a different one of the data drives; a check matrix comprising more than two rows of the at least one block of the check data in the main memory, each of the rows being stored on a different one of the check drives, one of the rows comprising a parity row comprising the Galois Field (GF) summation of all of the rows of the data matrix; a thread that executes on the SIMD CPU core and comprising: at least one parallel multiplier that multiplies the at least one vector of the data matrix by a single factor to compute parallel multiplier results comprising at least one vector; at least one parallel adder that adds the at least one vector of the parallel multiplier results and computes a running total; and a sequencer wherein the sequencer orders load operations of the original data into at least one of the vector registers and computes the check data with the parallel lookup multiplier and the parallel adder, and stores the computed check data from the vector registers to the main memory.

A. CLOUDERA’S DIRECT INFRINGEMENT

177. As to Cloudera, at least the Cloudera Infringing Products and Services, as defined above, comprise hardware and software components that together practice every element of one or more claims of the ’358 Patent, including at least Claim 1. The Cloudera Infringing Products and Services are systems adapted to use accelerated ECC processing to improve the storage and retrieval of digital data that is distributed across multiple drives. They comprise a processing core comprising a single-instruction-multiple-data (“SIMD”) central processing unit (“CPU”) core that executes the SIMD instructions and loads data from main memory and stores data to main memory. The SIMD CPU core, including for example Intel, AMD, ARM, and/or PPC64 processing cores, includes at least 16 data registers of at least 16 bytes each. The Cloudera Infringing Products and Services include a system drive with non-volatile storage (memory) for storing the SIMD computer instructions. The Cloudera Infringing Products and Services include multiple data drives, each of

which includes a memory that stores blocks of original data that are at least 512 bytes. The Cloudera Infringing Products and Services include more than two check drives, each of which includes a memory that stores blocks of check data. The Cloudera Infringing Products and Services further include an input/output (I/O) controller to coordinate communication and data transfers between the main memory and the non-volatile storage media and that stores the check data from the main memory to the check drives. The processing core, SIMD instructions, memory, and I/O controller of the Cloudera Infringing Products and Services implement accelerated ECC. The accelerated ECC system of the Cloudera Infringing Products and Services includes a data matrix for holding vectors of original data, with each row of a block of original data stored on a different data drive. The accelerated ECC system of the Cloudera Infringing Products and Services includes a check matrix for holding vectors of check data, with each row of a block of check data stored on different check drives. Moreover, one of the rows of the block of check data comprises a parity row comprising the Galois Field (GF) summation of all of the rows of the data matrix. The Cloudera Infringing Products and Services include a thread for executing on the SIMD CPU processing core that includes a parallel lookup multiplier, a parallel adder, and a sequencer. The parallel lookup multiplier of the Cloudera Infringing Products and Services multiplies a vector of the data matrix by a single factor; the parallel adder adds the result of the parallel multiplier to compute a running total; and the sequencer orders load operations of the data into the registers, computes the check data, and stores the computed check data to main memory.

B. ADP'S DIRECT INFRINGEMENT

178. As to ADP, at least the ADP Infringing Products and Services, as defined above, comprise hardware and software components that together practice every element of one or more claims of the '358 Patent, including at least Claim 1. The ADP Infringing Products and Services are systems adapted to use accelerated ECC processing to improve the storage and retrieval of

digital data that is distributed across multiple drives. They comprise a processing core comprising a single-instruction-multiple-data (“SIMD”) central processing unit (“CPU”) core that executes the SIMD instructions and loads data from main memory and stores data to main memory. The SIMD CPU core, including for example Intel, AMD, ARM, and/or PPC64 processing cores, includes at least 16 data registers of at least 16 bytes each. The ADP Infringing Products and Services include a system drive with non-volatile storage (memory) for storing the SIMD computer instructions. The ADP Infringing Products and Services include multiple data drives, each of which includes a memory that stores blocks of original data that are at least 512 bytes. The ADP Infringing Products and Services include more than two check drives, each of which includes a memory that stores blocks of check data. The ADP Infringing Products and Services further include an input/output (I/O) controller to coordinate communication and data transfers between the main memory and the non-volatile storage media and that stores the check data from the main memory to the check drives. The processing core, SIMD instructions, memory, and I/O controller of the ADP Infringing Products and Services implement accelerated ECC. The accelerated ECC system of the ADP Infringing Products and Services includes a data matrix for holding vectors of original data, with each row of a block of original data stored on a different data drive. The accelerated ECC system of the ADP Infringing Products and Services includes a check matrix for holding vectors of check data, with each row of a block of check data stored on different check drives. Moreover, one of the rows of the block of check data comprises a parity row comprising the Galois Field (GF) summation of all of the rows of the data matrix. The ADP Infringing Products and Services include a thread for executing on the SIMD CPU processing core that includes a parallel lookup multiplier, a parallel adder, and a sequencer. The parallel lookup multiplier of the ADP Infringing Products and Services multiplies a vector of the data matrix by a

single factor; the parallel adder adds the result of the parallel multiplier to compute a running total; and the sequencer orders load operations of the data into the registers, computes the check data, and stores the computed check data to main memory.

C. EXPERIAN'S DIRECT INFRINGEMENT

179. As to Experian, at least the Experian Infringing Products and Services, as defined above, comprise hardware and software components that together practice every element of one or more claims of the '358 Patent, including at least Claim 1. The Experian Infringing Products and Services are systems adapted to use accelerated ECC processing to improve the storage and retrieval of digital data that is distributed across multiple drives. They comprise a processing core comprising a single-instruction-multiple-data ("SIMD") central processing unit ("CPU") core that executes the SIMD instructions and loads data from main memory and stores data to main memory. The SIMD CPU core, including for example Intel, AMD, ARM, and/or PPC64 processing cores, includes at least 16 data registers of at least 16 bytes each. The Experian Infringing Products and Services include a system drive with non-volatile storage (memory) for storing the SIMD computer instructions. The Experian Infringing Products and Services include multiple data drives, each of which includes a memory that stores blocks of original data that are at least 512 bytes. The Experian Infringing Products and Services include more than two check drives, each of which includes a memory that stores blocks of check data. The Experian Infringing Products and Services further include an input/output (I/O) controller to coordinate communication and data transfers between the main memory and the non-volatile storage media and that stores the check data from the main memory to the check drives. The processing core, SIMD instructions, memory, and I/O controller of the Experian Infringing Products and Services implement accelerated ECC. The accelerated ECC system of the Experian Infringing Products and Services includes a data matrix for holding vectors of original data, with each row of a block of original data stored on a different

data drive. The accelerated ECC system of the Experian Infringing Products and Services includes a check matrix for holding vectors of check data, with each row of a block of check data stored on different check drives. Moreover, one of the rows of the block of check data comprises a parity row comprising the Galois Field (GF) summation of all of the rows of the data matrix. The Experian Infringing Products and Services include a thread for executing on the SIMD CPU processing core that includes a parallel lookup multiplier, a parallel adder, and a sequencer. The parallel lookup multiplier of the Experian Infringing Products and Services multiplies a vector of the data matrix by a single factor; the parallel adder adds the result of the parallel multiplier to compute a running total; and the sequencer orders load operations of the data into the registers, computes the check data, and stores the computed check data to main memory.

D. WARGAMING'S DIRECT INFRINGEMENT

180. As to Wargaming, at least the Wargaming Infringing Products and Services, as defined above, comprise hardware and software components that together practice every element of one or more claims of the '358 Patent, including at least Claim 1. The Wargaming Infringing Products and Services are systems adapted to use accelerated ECC processing to improve the storage and retrieval of digital data that is distributed across multiple drives. They comprise a processing core comprising a single-instruction-multiple-data ("SIMD") central processing unit ("CPU") core that executes the SIMD instructions and loads data from main memory and stores data to main memory. The SIMD CPU core, including for example Intel, AMD, ARM, and/or PPC64 processing cores, includes at least 16 data registers of at least 16 bytes each. The Wargaming Infringing Products and Services include a system drive with non-volatile storage (memory) for storing the SIMD computer instructions. The Wargaming Infringing Products and Services include multiple data drives, each of which includes a memory that stores blocks of original data that are at least 512 bytes. The Wargaming Infringing Products and Services include

more than two check drives, each of which includes a memory that stores blocks of check data. The Wargaming Infringing Products and Services further include an input/output (I/O) controller to coordinate communication and data transfers between the main memory and the non-volatile storage media and that stores the check data from the main memory to the check drives. The processing core, SIMD instructions, memory, and I/O controller of the Wargaming Infringing Products and Services implement accelerated ECC. The accelerated ECC system of the Wargaming Infringing Products and Services includes a data matrix for holding vectors of original data, with each row of a block of original data stored on a different data drive. The accelerated ECC system of the Wargaming Infringing Products and Services includes a check matrix for holding vectors of check data, with each row of a block of check data stored on different check drives. Moreover, one of the rows of the block of check data comprises a parity row comprising the Galois Field (GF) summation of all of the rows of the data matrix. The Wargaming Infringing Products and Services include a thread for executing on the SIMD CPU processing core that includes a parallel lookup multiplier, a parallel adder, and a sequencer. The parallel lookup multiplier of the Wargaming Infringing Products and Services multiplies a vector of the data matrix by a single factor; the parallel adder adds the result of the parallel multiplier to compute a running total; and the sequencer orders load operations of the data into the registers, computes the check data, and stores the computed check data to main memory.

II. INDIRECT INFRINGEMENT

181. In violation of 35 U.S.C. §§ 271(b), Intel is and has been infringing one or more of the '358 Patent's claims, including at least Claim 1, indirectly by inducing the infringement of at least Claim 1 of the '358 Patent by third parties, including for example Cloudera, ADP, Experian, and Wargaming, in this District and elsewhere in the United States. Direct infringement is the result of activities performed by users of systems that incorporate, among other features, ISA-L,

including for example Cloudera, ADP, Experian, and Wargaming, in accordance with at least Claim 1 of the '358 Patent.

182. Intel's affirmative acts of selling and/or distributing ISA-L (or portions thereof), causing ISA-L (or portions thereof) to be manufactured and distributed, providing instructive materials and information concerning operation and use of ISA-L (or portions thereof), and providing maintenance/service for such products or services, induced Cloudera, ADP, Experian, and Wargaming to infringe at least Claim 1 of the '358 Patent. For example, Intel induced Cloudera, ADP, Experian, and Wargaming to infringe at least Claim 1 of the '358 Patent through the implementation of ISA-L in the Cloudera, ADP, Experian, and Wargaming Infringing Products and Services. By and through these acts, Intel knowingly and specifically intended the users of ISA-L (or portions thereof) to infringe at least Claim 1 of the '358 Patent. Intel (1) knew or should have known of the '358 Patent since at least 2018, (2) performed and continues to perform affirmative acts that constitute induced infringement, and (3) knew or should have known that those acts would induce actual infringement of one or more of the '358 Patent's claims by users of ISA-L.

183. For example, upon information and belief, Intel (i) maintains a website to promote ISA-L,²¹ including to the EC System Defendants, (ii) produces videos regarding ISA-L and its use that are available to the EC System Defendants on the Intel website,²² (iii) describes case studies on big data optimization using ISA-L that are available to the EC System Defendants on the Intel

²¹ *E.g.*, Intel, Intel® Intelligent Storage Acceleration Library, *available at* <https://software.intel.com/content/www/us/en/develop/tools/isa-l.html> (last visited May 24, 2021).

²² *See, e.g.*, Intel, Erasure Code and Intel® Intelligent Storage Acceleration Library (Intel® ISA-L, *available at* <https://www.intel.com/content/www/us/en/products/docs/storage/erasure-code-isa-l-solution-video.html> (last visited May 24, 2021).

website, (iv) hosts articles, blog posts, and webinars regarding the use of ISA-L that are available to the EC System Defendants on the Intel website, and (v) publishes and makes available an API Reference Manual for ISA-L²³ that is available to the EC System Defendants, which it updates regularly.²⁴ Upon information and belief, Intel further offers the EC System Defendants technical support for ISA-L and the EC System Defendants' products.

184. Upon information and belief, Intel promotes and encourages the EC System Defendants to use ISA-L in order to drive sales of other Intel products and services to the EC System Defendants.

185. As to Intel, at least ISA-L, as defined above, is designed to be used with other components that, when combined with hardware, practice one or more claims of the '358 Patent, including at least Claim 1. EC Systems that employ ISA-L create a data matrix for holding vectors of original data, with each row of a block of original data stored on a different data drive. The systems that employ ISA-L create a check matrix for holding vectors of check data, with each row of a block of check data stored on different check drives. Moreover, one of the rows of the block of check data comprises a parity row comprising the Galois Field (GF) summation of all of the rows of the data matrix. The systems also include a thread for executing on the SIMD CPU processing core that includes a parallel lookup multiplier, a parallel adder, and a sequencer. The systems' parallel lookup multiplier multiplies a vector of the data matrix by a single factor; the

²³ See, e.g., Intel, Intel® Intelligent Storage Acceleration Library (Intel® ISA-L) Open Source Version, API Reference Manual (ver. 2.8, Sept. 27, 2013), available at https://01.org/sites/default/files/documentation/isa-l_open_src_2.8_0.pdf (last visited May 24, 2021).

²⁴ See, e.g., Intel, Intel® Intelligent Storage Acceleration Library (Intel® ISA-L), API Reference Manual (ver. 2.23.0, June 29, 2018), available at https://01.org/sites/default/files/documentation/isa-l_api_2.23.0.pdf (last visited May 24, 2021).

systems' parallel adder adds the result of the parallel multiplier to compute a running total; and the systems' sequencer orders load operations of the data into the registers, computes the check data, and stores the computed check data to main memory.

186. Especially in light of its actual knowledge of StreamScale and StreamScale's patent portfolio, Intel subjectively believed there was a high probability that StreamScale's Patents-in-Suit implicated ISA-L and that EC System Defendants use of ISA-L would infringe StreamScale's Patents-in-Suit, including the '358 Patent. To the extent that Intel lacked actual knowledge of the '358 Patent or the EC System Defendants' actual infringement of the '358 Patent, Intel took deliberate actions to avoid learning of those facts. Indeed, Intel actively encouraged others to ignore StreamScale and its patents and further reprimanded at least one employee for failing to ignore StreamScale and its patents.

187. At a minimum, Intel has had actual notice of the '358 Patent since March 5, 2021 and has knowledge of the infringing nature of its activities, yet continues to induce infringement of at least Claim 1 of the '358 Patent by Cloudera, ADP, Experian, and Wargaming.

188. Despite knowing of the '358 Patent since at least as early as March 5, 2021, upon information and belief, Intel has never undertaken any serious investigation to form a good faith belief as to non-infringement or invalidity of the '358 Patent.

189. Despite knowing of the '358 Patent since at least as early as March 5, 2021, Intel has continued to infringe one or more claims of the '358 Patent.

190. Despite knowing of the '358 Patent since at least July 7, 2021, Intel has continued to infringe one or more claims of the '358 Patent.

191. Therefore, upon information and belief, Intel's infringement of at least Claim 1 of the '358 Patent has been and continues to be willful, wanton, malicious, bad-faith, deliberate,

consciously wrongful, flagrant, or characteristic of a pirate, entitling StreamScale to increased damages pursuant to 35 U.S.C. § 284 and to attorneys' fees and costs incurred in prosecuting this action pursuant to 35 U.S.C. § 285.

III. DAMAGES

192. Defendants' acts of infringement have caused damages to StreamScale, and StreamScale is entitled to recover from Defendants the damages sustained by StreamScale as a result of Defendants' wrongful acts in an amount to be determined at trial.

COUNT 5—INFRINGEMENT OF THE '259 PATENT

193. StreamScale incorporates by reference the allegations set forth in Paragraphs 1–192 of this Complaint as though fully set forth herein.

I. DIRECT INFRINGEMENT

194. In violation of 35 U.S.C. § 271(a), Cloudera, ADP, Experian, and Wargaming are and have been directly infringing one or more of the '259 Patent's claims, including at least Claim 1, by making, using, selling, and/or offering for sale in the United States, and/or importing into the United States, without authority, erasure code products and services, including but not limited to those utilizing ISA-L, including without limitation the Cloudera Infringing Products and Services, the ADP Infringing Products and Services, the Experian Infringing Products and Services, and the Wargaming Infringing Products and Services, as described above.

195. The EC System Defendants are infringing claims of the '259 Patent, including at least Claim 1, literally and/or pursuant to the doctrine of equivalents.

196. Claim 1 of the '259 Patent is directed to a system adapted to use accelerated error-correcting code (ECC) processing to improve the storage and retrieval of digital data distributed across a plurality of drives, comprising: at least one processor comprising at least one single-instruction-multiple-data (SIMD) central processing unit (CPU) core that executes SIMD

instructions and loads original data from a main memory and stores check data to the main memory, the SIMD CPU core comprising at least 16 vector registers, each of the vector registers storing at least 16 bytes; at least one system drive comprising at least one non-volatile storage medium that stores the SIMD instructions; a plurality of data drives each comprising at least one non-volatile storage medium that stores at least one block of the original data, the at least one block comprising at least 512 bytes; more than two check drives each comprising at least one non-volatile storage medium that stores at least one block of the check data; at least one first input/output (I/O) controller that receives the at least one block of the original data from a transmitter and that stores the at least one block of the original data to the main memory; and at least one second input/output (I/O) controller that stores the at least one block of the check data from the main memory to the check drives, wherein the processor, the SIMD instructions, the non-volatile storage medium, and the at least one second I/O controller are configured to implement an erasure coding system comprising: a data matrix comprising at least one vector and comprising a plurality of rows of at least one block of the original data in the main memory, each of the rows being stored on a different one of the data drives; a check matrix comprising more than two rows of the at least one block of the check data in the main memory, each of the rows being stored on a different one of the check drives, one of the rows comprising a parity row comprising the Galois Field (GF) summation of all of the rows of the data matrix; and a thread that executes on the SIMD CPU core and comprising: at least one parallel multiplier that multiplies the at least one vector of the data matrix by a single factor to compute parallel multiplier results comprising at least one vector; at least one parallel adder that adds the at least one vector of the parallel multiplier results and computes a running total; and a sequencer wherein the sequencer orders load operations of the original data into at least one of the vector registers and computes the check data with the parallel multiplier

and the parallel adder, and stores the computed check data from the vector registers to the main memory.

A. CLOUDERA'S DIRECT INFRINGEMENT

197. As to Cloudera, at least the Cloudera Infringing Products and Services, as defined above, comprise hardware and software components that together practice every element of one or more claims of the '259 Patent, including at least Claim 1. The Cloudera Infringing Products and Services are systems adapted to use accelerated ECC processing to improve the storage and retrieval of digital data that is distributed across multiple drives. They comprise a processing core comprising a single-instruction-multiple-data ("SIMD") central processing unit ("CPU") core that executes the SIMD instructions and loads data from main memory and stores data to main memory. The SIMD CPU core, including for example Intel, AMD, ARM, and/or PPC64 processing cores, includes at least 16 data registers of at least 16 bytes each. The Cloudera Infringing Products and Services include a system drive with non-volatile storage (memory) for storing the SIMD computer instructions. The Cloudera Infringing Products and Services include multiple data drives, each of which includes a memory that stores blocks of original data that are at least 512 bytes. The Cloudera Infringing Products and Services include more than two check drives, each of which includes a memory that stores blocks of check data. The Cloudera Infringing Products and Services further include a first input/output (I/O) controller to receive blocks of original data from a transmitter and store that data to the main memory. The Cloudera Infringing Products and Services further include a second I/O controller to store blocks of check data from the main memory to the check drives. The processing core, SIMD instructions, memory, and I/O controller of the Cloudera Infringing Products and Services implement accelerated ECC. The accelerated ECC system of the Cloudera Infringing Products and Services includes a data matrix for holding vectors of original data, with each row of a block of original data stored on a different data drive.

The accelerated ECC system of the Cloudera Infringing Products and Services includes a check matrix for holding vectors of check data, with each row of a block of check data stored on different check drives. Moreover, one of the rows of the block of check data comprises a parity row comprising the Galois Field (GF) summation of all of the rows of the data matrix. The Cloudera Infringing Products and Services include a thread for executing on the SIMD CPU processing core that includes a parallel lookup multiplier, a parallel adder, and a sequencer. The parallel lookup multiplier of the Cloudera Infringing Products and Services multiplies a vector of the data matrix by a single factor; the parallel adder adds the result of the parallel multiplier to compute a running total; and the sequencer orders load operations of the data into the registers, computes the check data, and stores the computed check data to main memory.

B. ADP'S DIRECT INFRINGEMENT

198. As to ADP, at least the ADP Infringing Products and Services, as defined above, comprise hardware and software components that together practice every element of one or more claims of the '259 Patent, including at least Claim 1. The ADP Infringing Products and Services are systems adapted to use accelerated ECC processing to improve the storage and retrieval of digital data that is distributed across multiple drives. They comprise a processing core comprising a single-instruction-multiple-data ("SIMD") central processing unit ("CPU") core that executes the SIMD instructions and loads data from main memory and stores data to main memory. The SIMD CPU core, including for example Intel, AMD, ARM, and/or PPC64 processing cores, includes at least 16 data registers of at least 16 bytes each. The ADP Infringing Products and Services include a system drive with non-volatile storage (memory) for storing the SIMD computer instructions. The ADP Infringing Products and Services include multiple data drives, each of which includes a memory that stores blocks of original data that are at least 512 bytes. The ADP Infringing Products and Services include more than two check drives, each of which includes a

memory that stores blocks of check data. The ADP Infringing Products and Services further include a first input/output (I/O) controller to receive blocks of original data from a transmitter and store that data to the main memory. The ADP Infringing Products and Services further include a second I/O controller to store blocks of check data from the main memory to the check drives. The processing core, SIMD instructions, memory, and I/O controller of the ADP Infringing Products and Services implement accelerated ECC. The accelerated ECC system of the ADP Infringing Products and Services includes a data matrix for holding vectors of original data, with each row of a block of original data stored on a different data drive. The accelerated ECC system of the ADP Infringing Products and Services includes a check matrix for holding vectors of check data, with each row of a block of check data stored on different check drives. Moreover, one of the rows of the block of check data comprises a parity row comprising the Galois Field (GF) summation of all of the rows of the data matrix. The ADP Infringing Products and Services include a thread for executing on the SIMD CPU processing core that includes a parallel lookup multiplier, a parallel adder, and a sequencer. The parallel lookup multiplier of the ADP Infringing Products and Services multiplies a vector of the data matrix by a single factor; the parallel adder adds the result of the parallel multiplier to compute a running total; and the sequencer orders load operations of the data into the registers, computes the check data, and stores the computed check data to main memory.

C. EXPERIAN'S DIRECT INFRINGEMENT

199. As to Experian, at least the Experian Infringing Products and Services, as defined above, comprise hardware and software components that together practice every element of one or more claims of the '259 Patent, including at least Claim 1. The Experian Infringing Products and Services are systems adapted to use accelerated ECC processing to improve the storage and retrieval of digital data that is distributed across multiple drives. They comprise a processing core

comprising a single-instruction-multiple-data (“SIMD”) central processing unit (“CPU”) core that executes the SIMD instructions and loads data from main memory and stores data to main memory. The SIMD CPU core, including for example Intel, AMD, ARM, and/or PPC64 processing cores, includes at least 16 data registers of at least 16 bytes each. The Experian Infringing Products and Services include a system drive with non-volatile storage (memory) for storing the SIMD computer instructions. The Experian Infringing Products and Services include multiple data drives, each of which includes a memory that stores blocks of original data that are at least 512 bytes. The Experian Infringing Products and Services include more than two check drives, each of which includes a memory that stores blocks of check data. The Experian Infringing Products and Services further include a first input/output (I/O) controller to receive blocks of original data from a transmitter and store that data to the main memory. The Experian Infringing Products and Services further include a second I/O controller to store blocks of check data from the main memory to the check drives. The processing core, SIMD instructions, memory, and I/O controller of the Experian Infringing Products and Services implement accelerated ECC. The accelerated ECC system of the Experian Infringing Products and Services includes a data matrix for holding vectors of original data, with each row of a block of original data stored on a different data drive. The accelerated ECC system of the Experian Infringing Products and Services includes a check matrix for holding vectors of check data, with each row of a block of check data stored on different check drives. Moreover, one of the rows of the block of check data comprises a parity row comprising the Galois Field (GF) summation of all of the rows of the data matrix. The Experian Infringing Products and Services include a thread for executing on the SIMD CPU processing core that includes a parallel lookup multiplier, a parallel adder, and a sequencer. The parallel lookup multiplier of the Experian Infringing Products and Services multiplies a vector of the data matrix by a single factor; the

parallel adder adds the result of the parallel multiplier to compute a running total; and the sequencer orders load operations of the data into the registers, computes the check data, and stores the computed check data to main memory.

D. WARGAMING'S DIRECT INFRINGEMENT

200. As to Wargaming, at least the Wargaming Infringing Products and Services, as defined above, comprise hardware and software components that together practice every element of one or more claims of the '259 Patent, including at least Claim 1. The Wargaming Infringing Products and Services are systems adapted to use accelerated ECC processing to improve the storage and retrieval of digital data that is distributed across multiple drives. They comprise a processing core comprising a single-instruction-multiple-data ("SIMD") central processing unit ("CPU") core that executes the SIMD instructions and loads data from main memory and stores data to main memory. The SIMD CPU core, including for example Intel, AMD, ARM, and/or PPC64 processing cores, includes at least 16 data registers of at least 16 bytes each. The Wargaming Infringing Products and Services include a system drive with non-volatile storage (memory) for storing the SIMD computer instructions. The Wargaming Infringing Products and Services include multiple data drives, each of which includes a memory that stores blocks of original data that are at least 512 bytes. The Wargaming Infringing Products and Services include more than two check drives, each of which includes a memory that stores blocks of check data. The Wargaming Infringing Products and Services further include a first input/output (I/O) controller to receive blocks of original data from a transmitter and store that data to the main memory. The Wargaming Infringing Products and Services further include a second I/O controller to store blocks of check data from the main memory to the check drives. The processing core, SIMD instructions, memory, and I/O controller of the Wargaming Infringing Products and Services implement accelerated ECC. The accelerated ECC system of the Wargaming Infringing

Products and Services includes a data matrix for holding vectors of original data, with each row of a block of original data stored on a different data drive. The accelerated ECC system of the Wargaming Infringing Products and Services includes a check matrix for holding vectors of check data, with each row of a block of check data stored on different check drives. Moreover, one of the rows of the block of check data comprises a parity row comprising the Galois Field (GF) summation of all of the rows of the data matrix. The Wargaming Infringing Products and Services include a thread for executing on the SIMD CPU processing core that includes a parallel lookup multiplier, a parallel adder, and a sequencer. The parallel lookup multiplier of the Wargaming Infringing Products and Services multiplies a vector of the data matrix by a single factor; the parallel adder adds the result of the parallel multiplier to compute a running total; and the sequencer orders load operations of the data into the registers, computes the check data, and stores the computed check data to main memory.

II. INDIRECT INFRINGEMENT

201. In violation of 35 U.S.C. §§ 271(b), Intel is and has been infringing one or more of the '259 Patent's claims, including at least Claim 1, indirectly by inducing the infringement of at least Claim 1 of the '259 Patent by third parties, including for example Cloudera, ADP, Experian, and Wargaming, in this District and elsewhere in the United States. Direct infringement is the result of activities performed by users of systems that incorporate, among other features, ISA-L, including for example Cloudera, ADP, Experian, and Wargaming, in accordance with at least Claim 1 of the '259 Patent.

202. Intel's affirmative acts of selling and/or distributing ISA-L (or portions thereof), causing ISA-L (or portions thereof) to be manufactured and distributed, providing instructive materials and information concerning operation and use of ISA-L (or portions thereof), and providing maintenance/service for such products or services, induced Cloudera, ADP, Experian,

and Wargaming to infringe at least Claim 1 of the '259 Patent. For example, Intel induced Cloudera, ADP, Experian, and Wargaming to infringe at least Claim 1 of the '259 Patent through the implementation of ISA-L in the Cloudera, ADP, Experian, and Wargaming Infringing Products and Services. By and through these acts, Intel knowingly and specifically intended the users of ISA-L (or portions thereof) to infringe at least Claim 1 of the '259 Patent. Intel (1) knew or should have known of the '259 Patent since at least 2019, (2) performed and continues to perform affirmative acts that constitute induced infringement, and (3) knew or should have known that those acts would induce actual infringement of one or more of the '259 Patent's claims by users of ISA-L.

203. For example, upon information and belief, Intel (i) maintains a website to promote ISA-L,²⁵ including to the EC System Defendants, (ii) produces videos regarding ISA-L and its use that are available to the EC System Defendants on the Intel website,²⁶ (iii) describes case studies on big data optimization using ISA-L that are available to the EC System Defendants on the Intel website, (iv) hosts articles, blog posts, and webinars regarding the use of ISA-L that are available to the EC System Defendants on the Intel website, and (v) publishes and makes available an API Reference Manual for ISA-L²⁷ that is available to the EC System Defendants, which it updates

²⁵ *E.g.*, Intel, Intel® Intelligent Storage Acceleration Library, *available at* <https://software.intel.com/content/www/us/en/develop/tools/isa-l.html> (last visited May 24, 2021).

²⁶ *See, e.g.*, Intel, Erasure Code and Intel® Intelligent Storage Acceleration Library (Intel® ISA-L, *available at* <https://www.intel.com/content/www/us/en/products/docs/storage/erasure-code-isa-l-solution-video.html> (last visited May 24, 2021).

²⁷ *See, e.g.*, Intel, Intel® Intelligent Storage Acceleration Library (Intel® ISA-L) Open Source Version, API Reference Manual (ver. 2.8, Sept. 27, 2013), *available at* https://01.org/sites/default/files/documentation/isa-l_open_src_2.8_0.pdf (last visited May 24, 2021).

regularly.²⁸ Upon information and belief, Intel further offers the EC System Defendants technical support for ISA-L and the EC System Defendants' products.

204. Upon information and belief, Intel promotes and encourages the EC System Defendants to use ISA-L in order to drive sales of other Intel products and services to the EC System Defendants.

205. As to Intel, at least ISA-L, as defined above, is designed to be used with other components that, when combined with hardware, practice one or more claims of the '259 Patent, including at least Claim 1. EC Systems that employ ISA-L create a data matrix for holding vectors of original data, with each row of a block of original data stored on a different data drive. The systems that employ ISA-L create a check matrix for holding vectors of check data, with each row of a block of check data stored on different check drives. Moreover, one of the rows of the block of check data comprises a parity row comprising the Galois Field (GF) summation of all of the rows of the data matrix. The systems also include a thread for executing on the SIMD CPU processing core that includes a parallel lookup multiplier, a parallel adder, and a sequencer. The systems' parallel lookup multiplier multiplies a vector of the data matrix by a single factor; the systems' parallel adder adds the result of the parallel multiplier to compute a running total; and the systems' sequencer orders load operations of the data into the registers, computes the check data, and stores the computed check data to main memory.

206. Especially in light of its actual knowledge of StreamScale and StreamScale's patent portfolio, Intel subjectively believed there was a high probability that StreamScale's Patents-in-Suit implicated ISA-L and that EC System Defendants use of ISA-L would infringe

²⁸ See, e.g., Intel, Intel® Intelligent Storage Acceleration Library (Intel® ISA-L), API Reference Manual (ver. 2.23.0, June 29, 2018), available at https://01.org/sites/default/files/documentation/isa-l_api_2.23.0.pdf (last visited May 24, 2021).

StreamScale's Patents-in-Suit, including the '259 Patent. To the extent that Intel lacked actual knowledge of the '259 Patent or the EC System Defendants' actual infringement of the '259 Patent, Intel took deliberate actions to avoid learning of those facts. Indeed, Intel actively encouraged others to ignore StreamScale and its patents and further reprimanded at least one employee for failing to ignore StreamScale and its patents.

207. At a minimum, Intel has had actual notice of the '259 Patent since March 5, 2021 and has knowledge of the infringing nature of its activities, yet continues to induce infringement of at least Claim 1 of the '259 Patent by Cloudera, ADP, Experian, and Wargaming.

208. Despite knowing of the '259 Patent since at least as early as March 5, 2021, upon information and belief, Intel has never undertaken any serious investigation to form a good faith belief as to non-infringement or invalidity of the '259 Patent.

209. Despite knowing of the '259 Patent since at least as early as March 5, 2021, Intel has continued to infringe one or more claims of the '259 Patent.

210. Despite knowing of the '259 Patent since at least July 7, 2021, Intel has continued to infringe one or more claims of the '259 Patent.

211. Therefore, upon information and belief, Intel's infringement of at least Claim 1 of the '259 Patent has been and continues to be willful, wanton, malicious, bad-faith, deliberate, consciously wrongful, flagrant, or characteristic of a pirate, entitling StreamScale to increased damages pursuant to 35 U.S.C. § 284 and to attorneys' fees and costs incurred in prosecuting this action pursuant to 35 U.S.C. § 285.

III. DAMAGES

212. Defendants' acts of infringement have caused damages to StreamScale, and StreamScale is entitled to recover from Defendants the damages sustained by StreamScale as a result of Defendants' wrongful acts in an amount to be determined at trial.

COUNT 6—INFRINGEMENT OF THE '10-296 PATENT

213. StreamScale incorporates by reference the allegations set forth in Paragraphs 1–212 of this Complaint as though fully set forth herein.

I. DIRECT INFRINGEMENT

214. In violation of 35 U.S.C. § 271(a), Cloudera, ADP, Experian, and Wargaming are and have been directly infringing one or more of the '10-296 Patent's claims, including at least Claim 1, by making, using, selling, and/or offering for sale in the United States, and/or importing into the United States, without authority, erasure code products and services, including but not limited to those utilizing ISA-L, including without limitation the Cloudera Infringing Products and Services, the ADP Infringing Products and Services, the Experian Infringing Products and Services, and the Wargaming Infringing Products and Services, as described above.

215. The EC System Defendants are infringing claims of the '10-296 Patent, including at least Claim 1, literally and/or pursuant to the doctrine of equivalents.

216. Claim 1 of the '10-296 Patent is directed to an accelerated error-correcting code (ECC) system operating across multiple drives, comprising: at least one processing circuit comprising a plurality of central processing unit (CPU) cores that executes CPU instructions and loads original data from a main memory and stores check data to the main memory, each of the CPU cores comprising at least 16 registers, and each of the registers storing at least 8 bytes; at least one system drive comprising at least one non-volatile storage medium that stores the CPU instructions; a plurality of data drives each comprising at least one non-volatile storage medium that stores at least one block of the original data; at least four check drives each comprising at least one non-volatile storage medium that stores at least one block of the check data corresponding to the at least one block of the original data; and at least one input/output (I/O) controller that receives the at least one block of the original data from a transmitter and that stores the at least one block

of the original data to a main memory; wherein the processing circuit, the CPU instructions, the main memory, the plurality of data drives, the at least four check drives, and the at least one I/O controller are configured to implement a multi-core erasure encoding system comprising: original data in the main memory comprised of the at least one block of original data from the plurality of data drives; check data in the main memory comprised of the at least one block of check data; an encoding matrix for holding first factors in the main memory, the first factors being for encoding the original data in the main memory into the check data in the main memory; and a scheduler for generating ECC data in parallel across a plurality of threads by: dividing the original data in the main memory into a plurality of data matrices; dividing the check data in the main memory into a plurality of check matrices; assigning corresponding ones of the data matrices and the check matrices in the main memory to the plurality of threads, wherein each thread comprises an encoder, the encoder comprising at least a portion of the encoding matrix, a Galois Field (GF) multiplier, a Galois Field (GF) adder, and a sequencer for ordering operations through at least one of the data matrices, corresponding ones of the check matrices, and the at least a portion of the encoding matrix in the main memory using the GF multiplier and the GF adder to generate the check data in the main memory; and assigning the plurality of threads to the plurality of CPU cores of the processing circuit to concurrently generate the check matrices in the main memory from corresponding ones of the data matrices in the main memory.

A. CLOUDERA'S DIRECT INFRINGEMENT

217. As to Cloudera, at least the Cloudera Infringing Products and Services, as defined above, comprise hardware and software components that together practice every element of one or more claims of the '10-296 Patent, including at least Claim 1. The Cloudera Infringing Products and Services are accelerated ECC systems operating across multiple drives. They comprise a processing circuit comprising multiple central processing unit ("CPU") cores that execute CPU

instructions and loads original data from main memory and stores check data to main memory. The CPU processing cores, including for example Intel, AMD, ARM, and/or PPC64 processing cores, each include at least 16 data registers of at least 8 bytes each. The Cloudera Infringing Products and Services include a system drive with non-volatile storage (memory) for storing the CPU instructions. The Cloudera Infringing Products and Services include multiple data drives, each of which includes a memory that stores blocks of original data. The Cloudera Infringing Products and Services include at least four check drives, each of which includes a memory that stores blocks of check data, each block of check data corresponding to a block of the original data. The Cloudera Infringing Products and Services further include an input/output (I/O) controller to receive blocks of original data from a transmitter and store that data to the main memory. The processing circuit, CPU instructions, memory, data drives, check drives, and I/O controller of the Cloudera Infringing Products and Services implement accelerated ECC. The accelerated ECC system of the Cloudera Infringing Products and Services includes original data blocks in main memory from the multiple data drives, check data blocks in main memory, and encoding matrix with first factors in main memory where the first factors are for encoding the original data into check data, and a scheduler that generates ECC data in parallel across multiple threads. The scheduler of the Cloudera Infringing Products and Services divides the original data in main memory into multiple data matrices and the check data in main memory into multiple check matrices. The scheduler of the Cloudera Infringing Products and Services further assigns data and check matrices to the threads. Each thread in the Cloudera Infringing Products and Services includes an encoder comprising at least part of the encoding matrix, a Galois Field (GF) multiplier, a GF adder, and a sequencer that orders operations of the data to generate the check data in the main memory. The scheduler of the Cloudera Infringing Products and Services further assigns the

threads to the various CPU cores of the processing circuit to concurrently generate the check matrices from the data matrices in main memory.

B. ADP'S DIRECT INFRINGEMENT

218. As to ADP, at least the ADP Infringing Products and Services, as defined above, comprise hardware and software components that together practice every element of one or more claims of the '10-296 Patent, including at least Claim 1. The ADP Infringing Products and Services are accelerated ECC systems operating across multiple drives. They comprise a processing circuit comprising multiple central processing unit ("CPU") cores that execute CPU instructions and loads original data from main memory and stores check data to main memory. The CPU processing cores, including for example Intel, AMD, ARM, and/or PPC64 processing cores, each include at least 16 data registers of at least 8 bytes each. The ADP Infringing Products and Services include a system drive with non-volatile storage (memory) for storing the CPU instructions. The ADP Infringing Products and Services include multiple data drives, each of which includes a memory that stores blocks of original data. The ADP Infringing Products and Services include at least four check drives, each of which includes a memory that stores blocks of check data, each block of check data corresponding to a block of the original data. The ADP Infringing Products and Services further include an input/output (I/O) controller to receive blocks of original data from a transmitter and store that data to the main memory. The processing circuit, CPU instructions, memory, data drives, check drives, and I/O controller of the ADP Infringing Products and Services implement accelerated ECC. The accelerated ECC system of the ADP Infringing Products and Services includes original data blocks in main memory from the multiple data drives, check data blocks in main memory, and encoding matrix with first factors in main memory where the first factors are for encoding the original data into check data, and a scheduler that generates ECC data in parallel across multiple threads. The scheduler of the ADP Infringing

Products and Services divides the original data in main memory into multiple data matrices and the check data in main memory into multiple check matrices. The scheduler of the ADP Infringing Products and Services further assigns data and check matrices to the threads. Each thread in the ADP Infringing Products and Services includes an encoder comprising at least part of the encoding matrix, a Galois Field (GF) multiplier, a GF adder, and a sequencer that orders operations of the data to generate the check data in the main memory. The scheduler of the ADP Infringing Products and Services further assigns the threads to the various CPU cores of the processing circuit to concurrently generate the check matrices from the data matrices in main memory.

C. EXPERIAN'S DIRECT INFRINGEMENT

219. As to Experian, at least the Experian Infringing Products and Services, as defined above, comprise hardware and software components that together practice every element of one or more claims of the '10-296 Patent, including at least Claim 1. The Experian Infringing Products and Services are accelerated ECC systems operating across multiple drives. They comprise a processing circuit comprising multiple central processing unit ("CPU") cores that execute CPU instructions and loads original data from main memory and stores check data to main memory. The CPU processing cores, including for example Intel, AMD, ARM, and/or PPC64 processing cores, each include at least 16 data registers of at least 8 bytes each. The Experian Infringing Products and Services include a system drive with non-volatile storage (memory) for storing the CPU instructions. The Experian Infringing Products and Services include multiple data drives, each of which includes a memory that stores blocks of original data. The Experian Infringing Products and Services include at least four check drives, each of which includes a memory that stores blocks of check data, each block of check data corresponding to a block of the original data. The Experian Infringing Products and Services further include an input/output (I/O) controller to receive blocks of original data from a transmitter and store that data to the main memory. The

processing circuit, CPU instructions, memory, data drives, check drives, and I/O controller of the Experian Infringing Products and Services implement accelerated ECC. The accelerated ECC system of the Experian Infringing Products and Services includes original data blocks in main memory from the multiple data drives, check data blocks in main memory, and encoding matrix with first factors in main memory where the first factors are for encoding the original data into check data, and a scheduler that generates ECC data in parallel across multiple threads. The scheduler of the Experian Infringing Products and Services divides the original data in main memory into multiple data matrices and the check data in main memory into multiple check matrices. The scheduler of the Experian Infringing Products and Services further assigns data and check matrices to the threads. Each thread in the Experian Infringing Products and Services includes an encoder comprising at least part of the encoding matrix, a Galois Field (GF) multiplier, a GF adder, and a sequencer that orders operations of the data to generate the check data in the main memory. The scheduler of the Experian Infringing Products and Services further assigns the threads to the various CPU cores of the processing circuit to concurrently generate the check matrices from the data matrices in main memory.

D. WARGAMING'S DIRECT INFRINGEMENT

220. As to Wargaming, at least the Wargaming Infringing Products and Services, as defined above, comprise hardware and software components that together practice every element of one or more claims of the '10-296 Patent, including at least Claim 1. The Wargaming Infringing Products and Services are accelerated ECC systems operating across multiple drives. They comprise a processing circuit comprising multiple central processing unit ("CPU") cores that execute CPU instructions and loads original data from main memory and stores check data to main memory. The CPU processing cores, including for example Intel, AMD, ARM, and/or PPC64 processing cores, each include at least 16 data registers of at least 8 bytes each. The Wargaming

Infringing Products and Services include a system drive with non-volatile storage (memory) for storing the CPU instructions. The Wargaming Infringing Products and Services include multiple data drives, each of which includes a memory that stores blocks of original data. The Wargaming Infringing Products and Services include at least four check drives, each of which includes a memory that stores blocks of check data, each block of check data corresponding to a block of the original data. The Wargaming Infringing Products and Services further include an input/output (I/O) controller to receive blocks of original data from a transmitter and store that data to the main memory. The processing circuit, CPU instructions, memory, data drives, check drives, and I/O controller of the Wargaming Infringing Products and Services implement accelerated ECC. The accelerated ECC system of the Wargaming Infringing Products and Services includes original data blocks in main memory from the multiple data drives, check data blocks in main memory, and encoding matrix with first factors in main memory where the first factors are for encoding the original data into check data, and a scheduler that generates ECC data in parallel across multiple threads. The scheduler of the Wargaming Infringing Products and Services divides the original data in main memory into multiple data matrices and the check data in main memory into multiple check matrices. The scheduler of the Wargaming Infringing Products and Services further assigns data and check matrices to the threads. Each thread in the Wargaming Infringing Products and Services includes an encoder comprising at least part of the encoding matrix, a Galois Field (GF) multiplier, a GF adder, and a sequencer that orders operations of the data to generate the check data in the main memory. The scheduler of the Wargaming Infringing Products and Services further assigns the threads to the various CPU cores of the processing circuit to concurrently generate the check matrices from the data matrices in main memory.

II. INDIRECT INFRINGEMENT

221. In violation of 35 U.S.C. §§ 271(b), Intel is and has been infringing one or more of the '10-296 Patent's claims, including at least Claim 1, indirectly by inducing the infringement of at least Claim 1 of the '10-296 Patent by third parties, including for example Cloudera, ADP, Experian, and Wargaming, in this District and elsewhere in the United States. Direct infringement is the result of activities performed by users of systems that incorporate, among other features, ISA-L, including for example Cloudera, ADP, Experian, and Wargaming, in accordance with at least Claim 1 of the '10-296 Patent.

222. Intel's affirmative acts of selling and/or distributing ISA-L (or portions thereof), causing ISA-L (or portions thereof) to be manufactured and distributed, providing instructive materials and information concerning operation and use of ISA-L (or portions thereof), and providing maintenance/service for such products or services, induced Cloudera, ADP, Experian, and Wargaming to infringe at least Claim 1 of the '10-296 Patent. For example, Intel induced Cloudera, ADP, Experian, and Wargaming to infringe at least Claim 1 of the '10-296 Patent through the implementation of ISA-L in the Cloudera, ADP, Experian, and Wargaming Infringing Products and Services. By and through these acts, Intel knowingly and specifically intended the users of ISA-L (or portions thereof) to infringe at least Claim 1 of the '10-296 Patent. Intel (1) knew or should have known of the '10-296 Patent since at least 2020, (2) performed and continues to perform affirmative acts that constitute induced infringement, and (3) knew or should have known that those acts would induce actual infringement of one or more of the '10-296 Patent's claims by users of ISA-L.

223. For example, upon information and belief, Intel (i) maintains a website to promote ISA-L,²⁹ including to the EC System Defendants, (ii) produces videos regarding ISA-L and its use that are available to the EC System Defendants on the Intel website,³⁰ (iii) describes case studies on big data optimization using ISA-L that are available to the EC System Defendants on the Intel website, (iv) hosts articles, blog posts, and webinars regarding the use of ISA-L that are available to the EC System Defendants on the Intel website, and (v) publishes and makes available an API Reference Manual for ISA-L³¹ that is available to the EC System Defendants, which it updates regularly.³² Upon information and belief, Intel further offers the EC System Defendants technical support for ISA-L and the EC System Defendants' products.

224. Upon information and belief, Intel promotes and encourages the EC System Defendants to use ISA-L in order to drive sales of other Intel products and services to the EC System Defendants.

225. As to Intel, at least ISA-L, as defined above, is designed to be used with other components that, when combined with hardware, practice one or more claims of the

²⁹ *E.g.*, Intel, Intel® Intelligent Storage Acceleration Library, *available at* <https://software.intel.com/content/www/us/en/develop/tools/isa-l.html> (last visited May 24, 2021).

³⁰ *See, e.g.*, Intel, Erasure Code and Intel® Intelligent Storage Acceleration Library (Intel® ISA-L), *available at* <https://www.intel.com/content/www/us/en/products/docs/storage/erasure-code-isa-l-solution-video.html> (last visited May 24, 2021).

³¹ *See, e.g.*, Intel, Intel® Intelligent Storage Acceleration Library (Intel® ISA-L) Open Source Version, API Reference Manual (ver. 2.8, Sept. 27, 2013), *available at* https://01.org/sites/default/files/documentation/isa-l_open_src_2.8_0.pdf (last visited May 24, 2021).

³² *See, e.g.*, Intel, Intel® Intelligent Storage Acceleration Library (Intel® ISA-L), API Reference Manual (ver. 2.23.0, June 29, 2018), *available at* https://01.org/sites/default/files/documentation/isa-l_api_2.23.0.pdf (last visited May 24, 2021).

'10-296 Patent, including at least Claim 1. EC Systems that employ ISA-L create original data blocks in main memory from the multiple data drives, check data blocks in main memory, and encoding matrix with first factors in main memory where the first factors are for encoding the original data into check data, and a scheduler that generates ECC data in parallel across multiple threads. The scheduler of the systems divides the original and check data in into multiple data and check matrices, respectively, and assigns data and check matrices to the threads. Each thread in the systems includes an encoder comprising at least part of the encoding matrix, a Galois Field (GF) multiplier, a GF adder, and a sequencer that orders operations of the data to generate the check data in the main memory.

226. Especially in light of its actual knowledge of StreamScale and StreamScale's patent portfolio, Intel subjectively believed there was a high probability that StreamScale's Patents-in-Suit implicated ISA-L and that EC System Defendants use of ISA-L would infringe StreamScale's Patents-in-Suit, including the '10-296 Patent. To the extent that Intel lacked actual knowledge of the '10-296 Patent or the EC System Defendants' actual infringement of the '10-296 Patent, Intel took deliberate actions to avoid learning of those facts. Indeed, Intel actively encouraged others to ignore StreamScale and its patents and further reprimanded at least one employee for failing to ignore StreamScale and its patents.

227. At a minimum, Intel has had actual notice of the '10-296 Patent since March 5, 2021 and has knowledge of the infringing nature of its activities, yet continues to induce infringement of at least Claim 1 of the '10-296 Patent by Cloudera, ADP, Experian, and Wargaming.

228. Despite knowing of the '10-296 Patent since at least as early as March 5, 2021, upon information and belief, Intel has never undertaken any serious investigation to form a good faith belief as to non-infringement or invalidity of the '10-296 Patent.

229. Despite knowing of the '10-296 Patent since at least as early as March 5, 2021, Intel has continued to infringe one or more claims of the '10-296 Patent.

230. Despite knowing of the '10-296 Patent since at least July 7, 2021, Intel has continued to infringe one or more claims of the '10-296 Patent.

231. Therefore, upon information and belief, Intel's infringement of at least Claim 1 of the '10-296 Patent has been and continues to be willful, wanton, malicious, bad-faith, deliberate, consciously wrongful, flagrant, or characteristic of a pirate, entitling StreamScale to increased damages pursuant to 35 U.S.C. § 284 and to attorneys' fees and costs incurred in prosecuting this action pursuant to 35 U.S.C. § 285.

III. DAMAGES

232. Defendants' acts of infringement have caused damages to StreamScale, and StreamScale is entitled to recover from Defendants the damages sustained by StreamScale as a result of Defendants' wrongful acts in an amount to be determined at trial.

DAMAGES

233. StreamScale is entitled to, and now seeks to, recover damages in an amount not less than the maximum amount permitted by law caused by Defendants' acts of infringement.

234. As a result of Defendants' acts of infringement, StreamScale has suffered actual and consequential damages. To the fullest extent permitted by law, StreamScale seeks recovery of damages in an amount to compensate for Defendants' infringement. StreamScale further seeks any other damages to which StreamScale would be entitled to in law or in equity.

INJUNCTIVE RELIEF

235. Defendants' acts of infringement have caused—and unless restrained and enjoined, Defendants' acts of infringement will continue to cause—irreparable injury and damage to StreamScale for which StreamScale has no adequate remedy at law. Unless preliminarily and permanently enjoined by this Court, Defendants will continue to infringe the Patents-in-Suit.

ATTORNEYS' FEES

236. StreamScale is entitled to recover reasonable and necessary attorneys' fees under applicable law.

DEMAND FOR JURY TRIAL

Pursuant to Rule 38 of the Federal Rules of Civil Procedure, StreamScale demands a trial by jury on all issues so triable.

PRAYER FOR RELIEF

StreamScale respectfully requests that the Court enter preliminary and final orders, declarations, and judgments against Defendants as are necessary to provide StreamScale with the following relief:

- a. A judgment that Defendants have infringed and/or are infringing one or more claims of the '8-296 Patent, literally or under the doctrine of equivalents, and directly or indirectly as alleged above;
- b. A judgment that Intel's infringement of the '8-296 Patent has been willful;
- c. A judgment that Defendants have infringed and/or are infringing one or more claims of the '374 Patent, literally or under the doctrine of equivalents, and directly or indirectly as alleged above;
- d. A judgment that Intel's infringement of the '374 Patent has been willful;

- e. A judgment that Defendants have infringed and/or are infringing one or more claims of the '759 Patent, literally or under the doctrine of equivalents, and directly or indirectly as alleged above;
- f. A judgment that Intel's infringement of the '759 Patent has been willful;
- g. A judgment that Defendants have infringed and/or are infringing one or more claims of the '358 Patent, literally or under the doctrine of equivalents, and directly or indirectly as alleged above;
- h. A judgment that Intel's infringement of the '358 Patent has been willful;
- i. A judgment that Defendants have infringed and/or are infringing one or more claims of the '259 Patent, literally or under the doctrine of equivalents, and directly or indirectly as alleged above;
- j. A judgment that Intel's infringement of the '259 Patent has been willful;
- k. A judgment that Defendants have infringed and/or are infringing one or more claims of the '10-296 Patent, literally or under the doctrine of equivalents, and directly or indirectly as alleged above;
- l. A judgment that Intel's infringement of the '10-296 Patent has been willful;
- m. An award for all damages arising out of Defendants' infringement, together with prejudgment and post-judgment interest, jointly and severally, in an amount according to proof, including without limitation attorneys' fees and litigation costs and expenses;
- n. An accounting of damages and any future compensation due to StreamScale for Defendants' infringement (past, present, or future) not specifically

accounted for in a damages award (or other relief), and/or permanent injunctive relief;

- o. An award of reasonable attorneys' fees as provided by 35 U.S.C. § 285 and enhanced damages as provided by 35 U.S.C. § 284;
- p. The entry of an order preliminarily and permanently enjoining and restraining Defendants and its parents, affiliates, subsidiaries, officers, agents, servants, employees, attorneys, successors, and assigns and all those person in active concert or participation with them or any of them, from making, importing, using, offering for sale, selling, or causing to be sold any product falling within the scope of any claim of the Patents-in-Suit, or otherwise infringing or inducing infringement of any claim of the Patents-in-Suit; and
- q. All further relief in law or in equity as the Court may deem just and proper.

Dated: July 9, 2021

Respectfully submitted,

/s/ Jamie H. McDole

Jamie H. McDole

State Bar No. 24082049

Phillip B. Philbin

State Bar No. 15909020

Michael D. Karson

State Bar No. 24090198

Nadia E. Haghghatian

State Bar No. 24087652

Austin C. Teng

State Bar No. 24093247

THOMPSON & KNIGHT LLP

One Arts Plaza

1722 Routh St., Suite 1500

Dallas, Texas 75201

Tel.: 214.969.1700

Fax: 214.969.1751

Email: jamie.mcdole@tklaw.com

phillip.philbin@tklaw.com

michael.karson@tklaw.com

nadia.haghghatian@tklaw.com

austin.teng@tklaw.com

Attorneys for Plaintiff StreamScale, Inc.

CERTIFICATE OF SERVICE

I hereby certify that, on July 9, 2021, I electronically submitted the foregoing document with the clerk of the United States District Court for the Western District of Texas, using the electronic case management CM/ECF system of the Court which will send notification of such filing to the following:

Brock S. Weber
Christopher Kao
Pillsbury Winthrop Shaw Pittman LLP
4 Embarcadero Center, 22nd Floor
San Francisco, CA 94111

Christopher S. Ponder
Harper Batts
Sheppard, Mullin, Richter & Hampton LLP
379 Lytton Avenue
Palo Alto, CA 94301

Steven P. Tepera
Pillsbury Winthrop Shaw Pittman LLP
401 Congress Avenue
Suite 1700
Austin, TX 78701

Jennifer Klein Ayers
Sheppard Mullin Richter & Hampton LLP
2200 Ross Avenue, 24th Floor
Dallas, TX 75201

*Counsel for Defendants Cloudera, Inc. &
Experian plc*

*Counsel for Defendant Wargaming (Austin),
Inc.*

Amanda L. Major
Wilmer Cutler Pickering Hale and Dorr LLP
1875 Pennsylvania Ave., NW
Washington, DC 20006

Jose Carlos Villarreal
Perkins Coie LLP
500 W. 2nd Street
Suite 1900
Austin, TX 78701

Annaleigh E. Curtis
Wilmer Cutler Pickering Hale and Dorr LLP
60 State Street
Boston, MA 02109

*Counsel for Defendant Automatic Data
Processing, Inc.*

Jennifer J. John
Sonal N. Mehta
Wilmer Cutler Pickering Hale and Dorr LLP
2600 El Camino Real, Suite 400
Palo Alto, CA 94306

Joseph Taylor Gooch
Wilmer Cutler Pickering Hale and Dorr LLP
One Front Street, Suite 3500
San Francisco, CA 94111

Vikram Iyer
Wilmer Cutler Pickering Hale and Dorr LLP
350 South Grand Avenue, Suite 2400
Los Angeles, CA 90071

Austin Michael Schnell
Brian Christopher Nash
Pillsbury Winthrop Shaw Pittman LLP
401 Congress Ave, Suite 1700
Austin, TX 78701

Counsel for Defendant Intel Corp.

/s/ Jamie H. McDole

Jamie H. McDole

EXHIBIT A



US008683296B2

(12) **United States Patent**
Anderson et al.

(10) **Patent No.:** **US 8,683,296 B2**
(45) **Date of Patent:** **Mar. 25, 2014**

(54) **ACCELERATED ERASURE CODING SYSTEM AND METHOD**

(75) Inventors: **Michael H. Anderson**, Los Angeles, CA (US); **Sarah Mann**, Tucson, AZ (US)

(73) Assignee: **Streamscale, Inc.**, Los Angeles, CA (US)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 313 days.

(21) Appl. No.: **13/341,833**

(22) Filed: **Dec. 30, 2011**

(65) **Prior Publication Data**

US 2013/0173996 A1 Jul. 4, 2013

(51) **Int. Cl.**
GI1C 29/00 (2006.01)
H03M 13/00 (2006.01)
G06F 11/00 (2006.01)

(52) **U.S. Cl.**
USPC **714/763**; 714/6.24; 714/752; 714/758;
714/768; 714/770; 714/773; 714/784; 714/786

(58) **Field of Classification Search**
USPC 714/6.24, 763, 752, 758, 768, 770, 773,
714/784, 786
See application file for complete search history.

(56) **References Cited**

U.S. PATENT DOCUMENTS

6,654,924	B1 *	11/2003	Hassner et al.	714/758
6,823,425	B2 *	11/2004	Ghosh et al.	711/114
7,350,126	B2 *	3/2008	Winograd et al.	714/752
7,930,337	B2	4/2011	Hasenplaugh et al.	
8,145,941	B2 *	3/2012	Jacobson	714/6.24
8,352,847	B2 *	1/2013	Gunnam	714/801
2011/0029756	A1 *	2/2011	Biscondi et al.	712/22
2012/0272036	A1 *	10/2012	Muralimanohar et al. ...	711/202

2013/0108048	A1 *	5/2013	Grube et al.	380/270
2013/0110962	A1 *	5/2013	Grube et al.	709/213
2013/0111552	A1 *	5/2013	Grube et al.	726/3
2013/0124932	A1 *	5/2013	Schuh et al.	714/718
2013/0173956	A1 *	7/2013	Anderson	714/6.24

OTHER PUBLICATIONS

Hafner et al., Matrix Methods for Lost Data Reconstruction in Erasure Codes, Nov. 16, 2005, USENIX FAST '05 Paper, pp. 1-26.*
Anvin; The mathematics of RAID-6; First Version Jan. 20, 2004; Last Updated Dec. 20, 2011; pp. 1-9.
Maddock, et al.; White Paper, Surviving Two Disk Failures Introducing Various "Raid 6" Implementations; Xyratex; pp. 1-13.
Plank; All About Erasure Codes:—Reed-Solomon Coding—LDPC Coding; Logistical Computing and Internetworking Laboratory, Department of Computer Science, University of Tennessee; ICL—Aug. 20, 2004; 52 sheets.

* cited by examiner

Primary Examiner — John J Tabone, Jr.

(74) Attorney, Agent, or Firm — Christie, Parker & Hale, LLP

(57) **ABSTRACT**

An accelerated erasure coding system includes a processing core for executing computer instructions and accessing data from a main memory, and a non-volatile storage medium for storing the computer instructions. The processing core, storage medium, and computer instructions are configured to implement an erasure coding system, which includes: a data matrix for holding original data in the main memory; a check matrix for holding check data in the main memory; an encoding matrix for holding first factors in the main memory, the first factors being for encoding the original data into the check data; and a thread for executing on the processing core. The thread includes: a parallel multiplier for concurrently multiplying multiple entries of the data matrix by a single entry of the encoding matrix; and a first sequencer for ordering operations through the data matrix and the encoding matrix using the parallel multiplier to generate the check data.

40 Claims, 9 Drawing Sheets

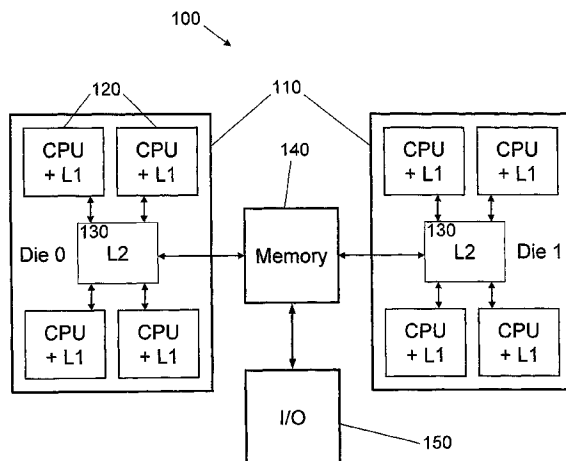


FIG. 1

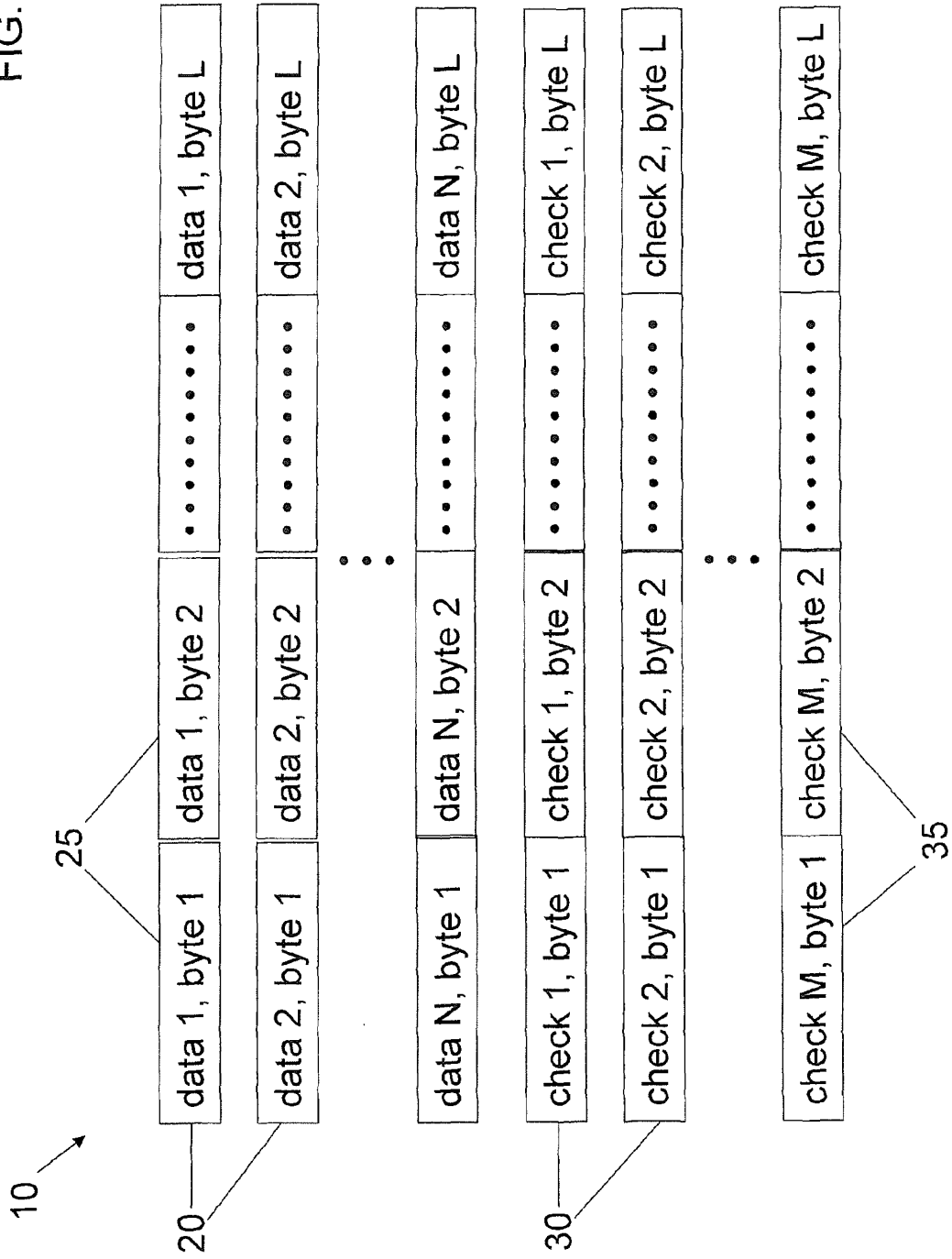


FIG. 2

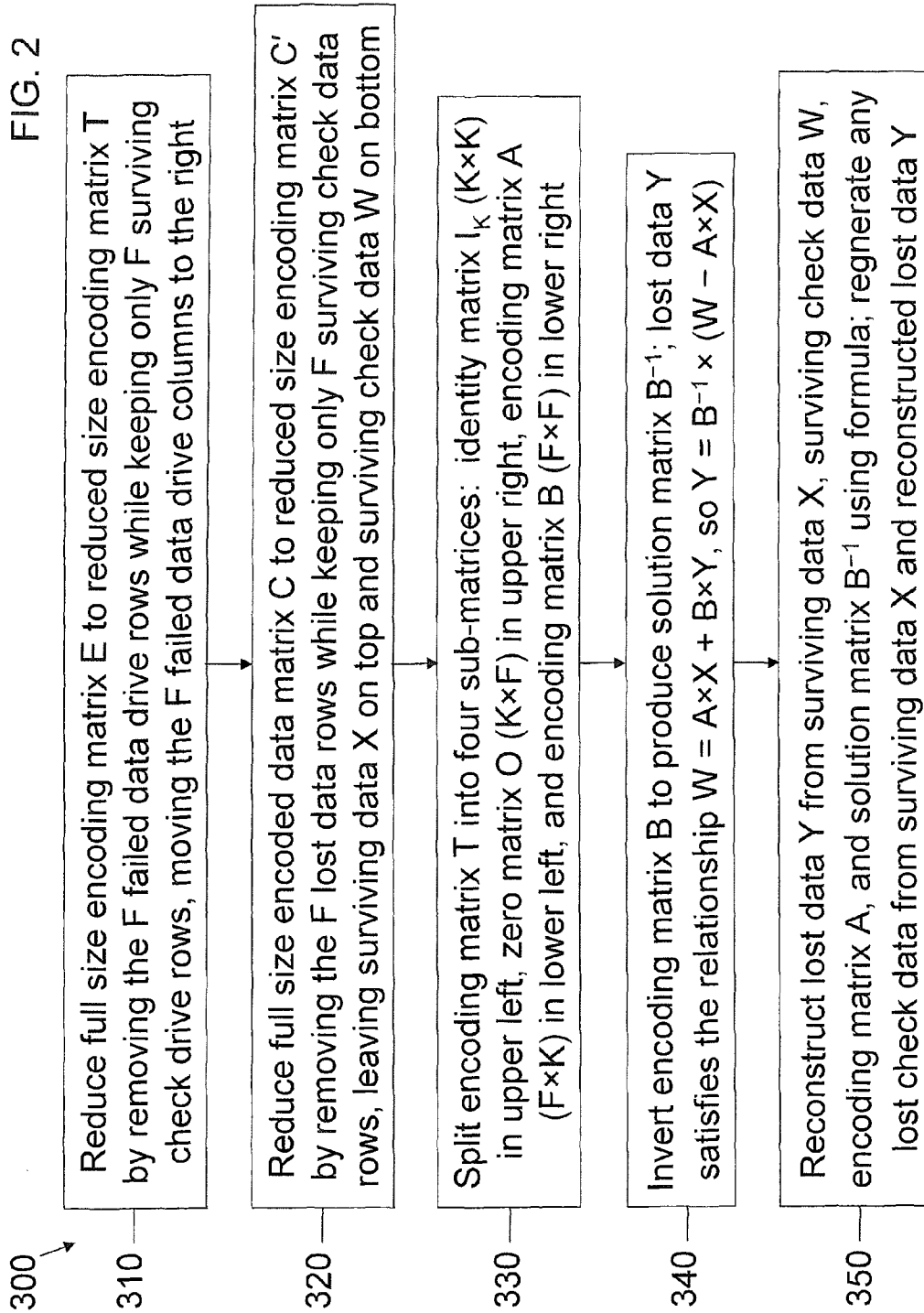


FIG. 3

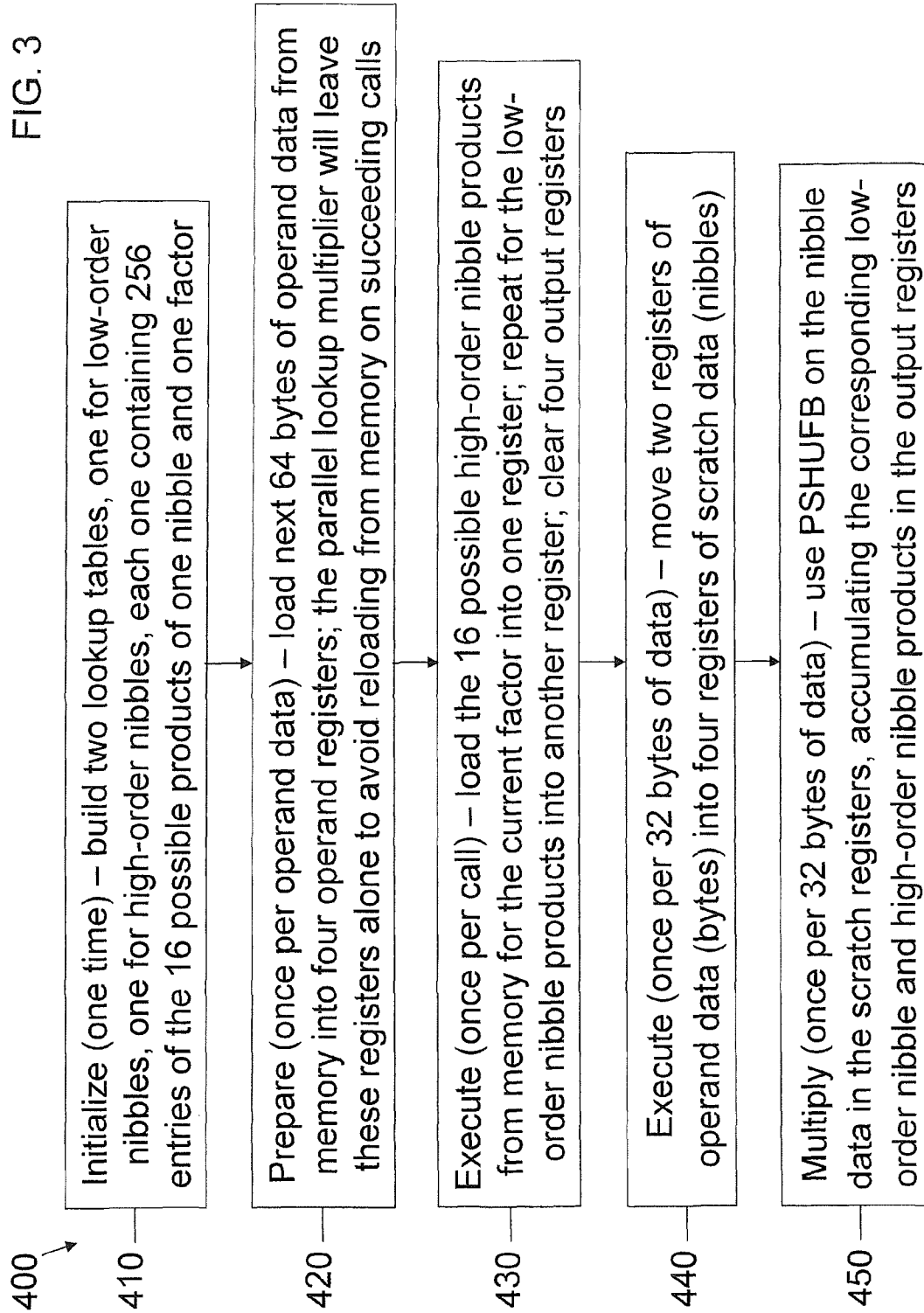


FIG. 4

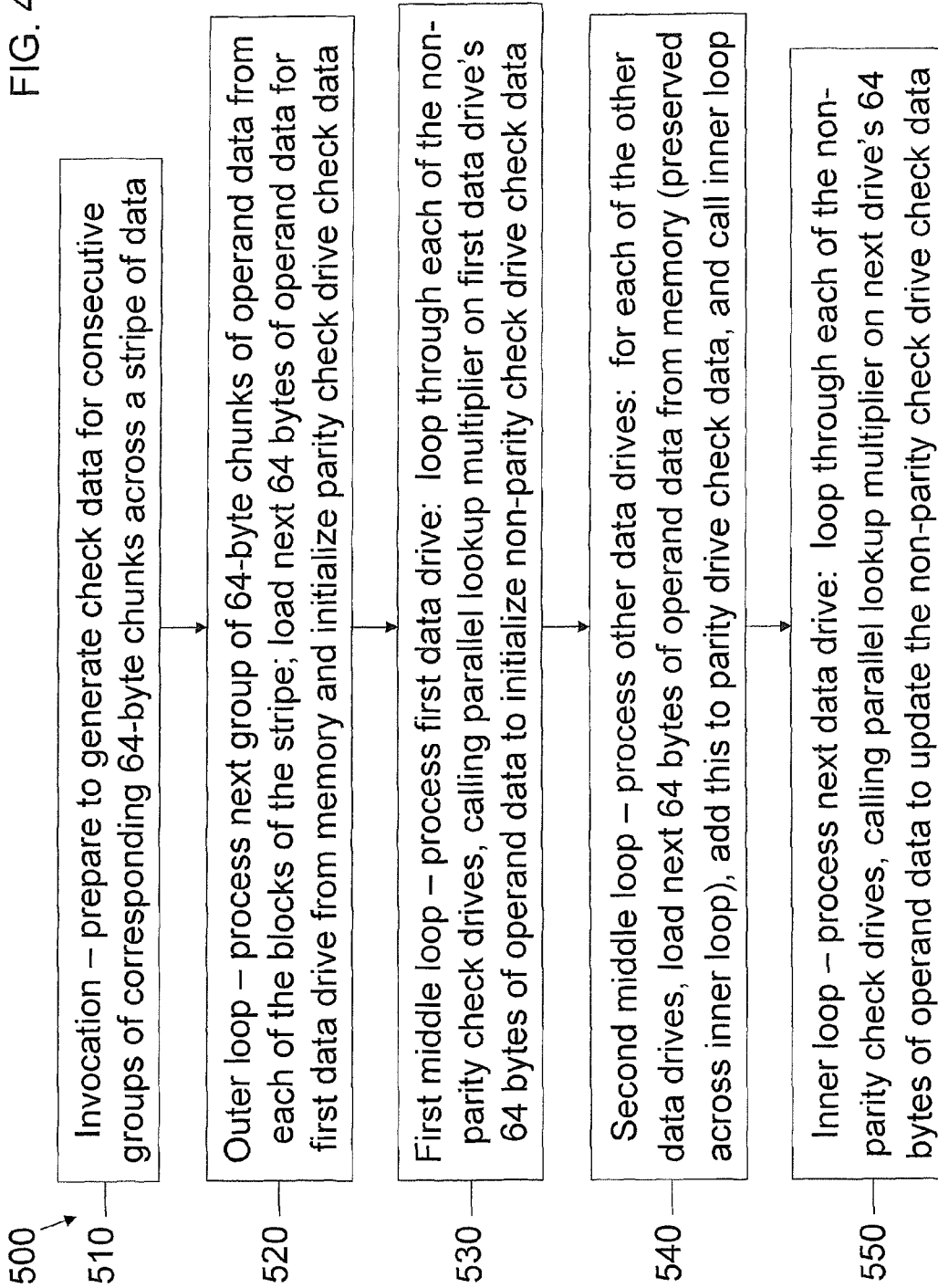


FIG. 5

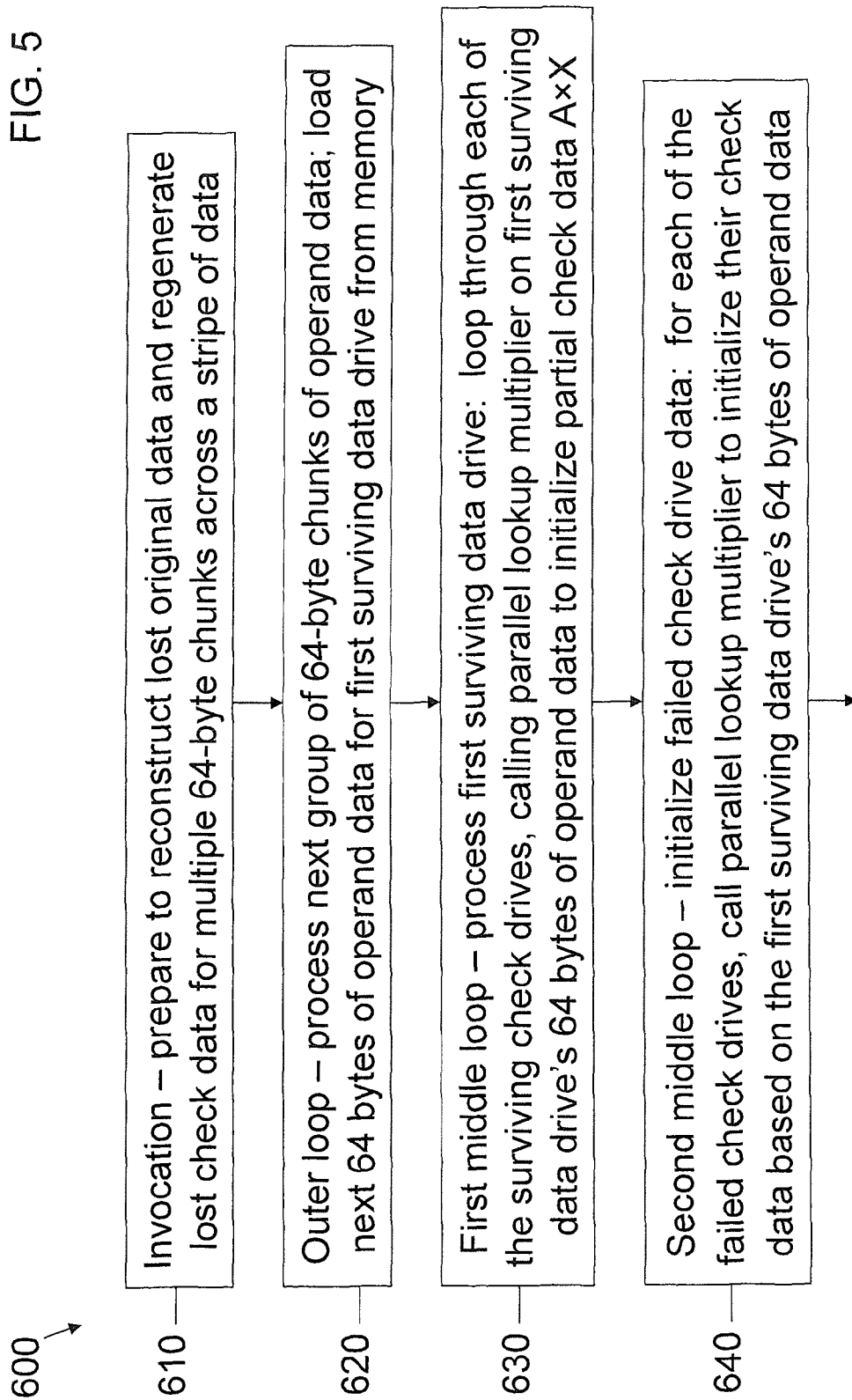


FIG. 6

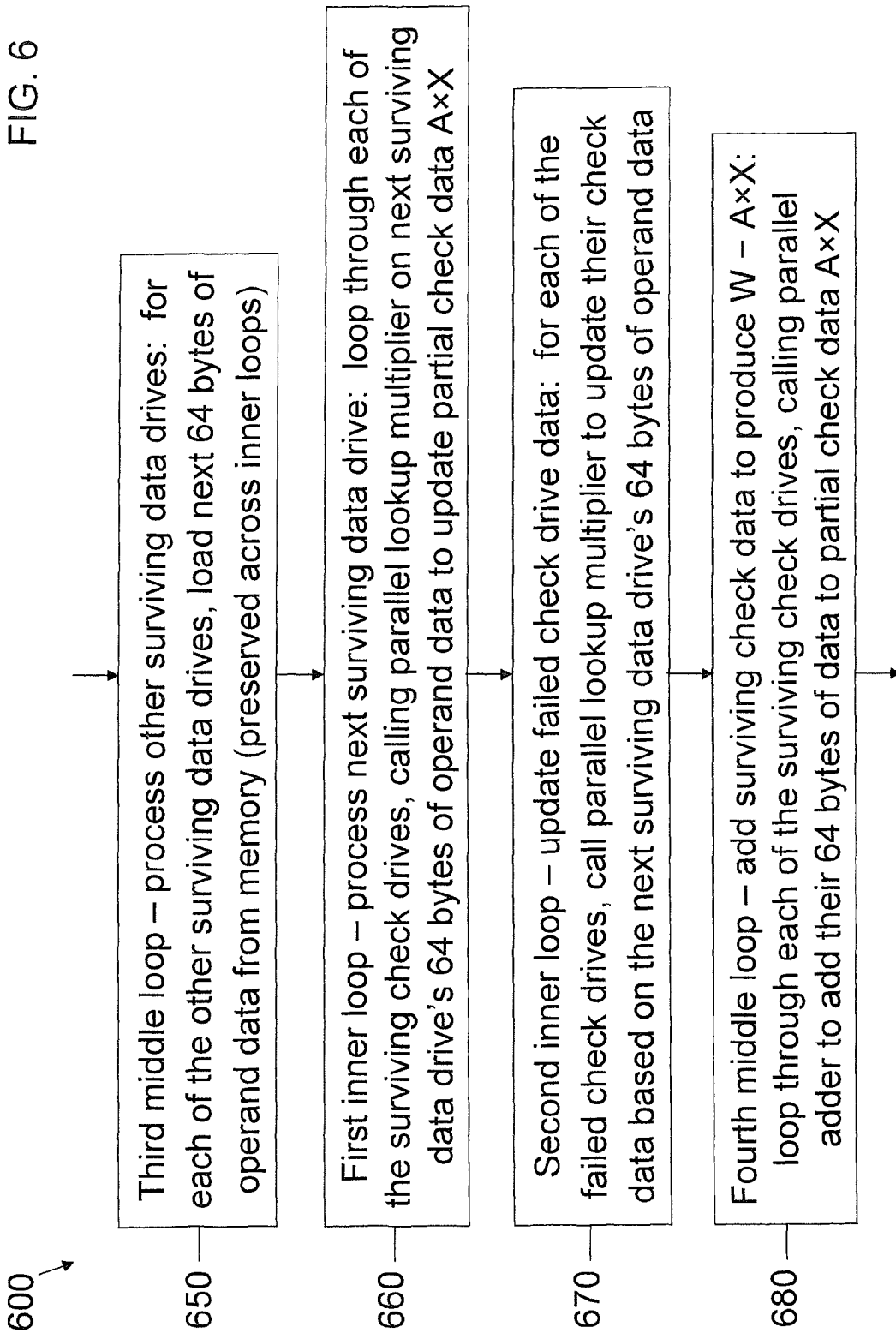


FIG. 7

600 →

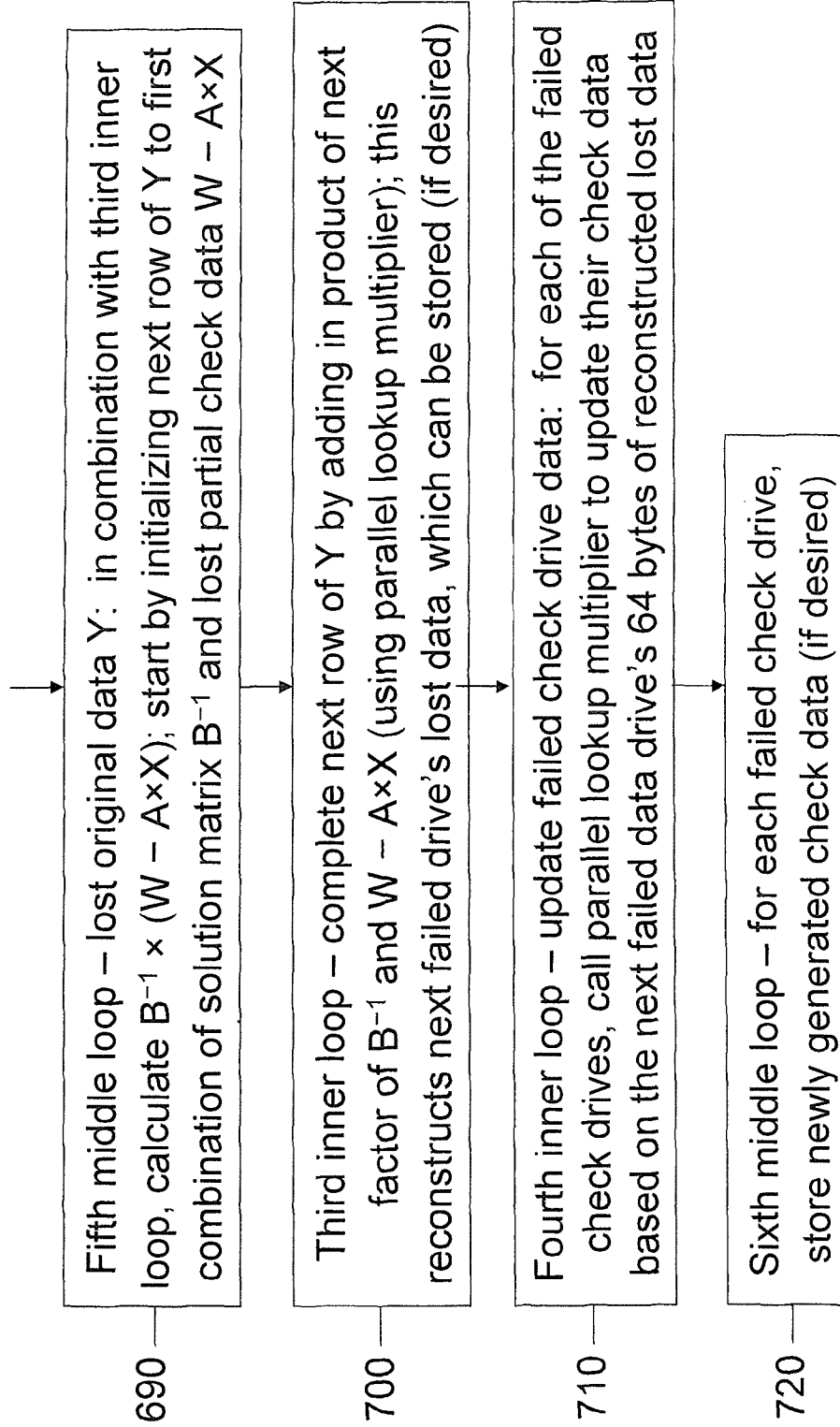


FIG. 8

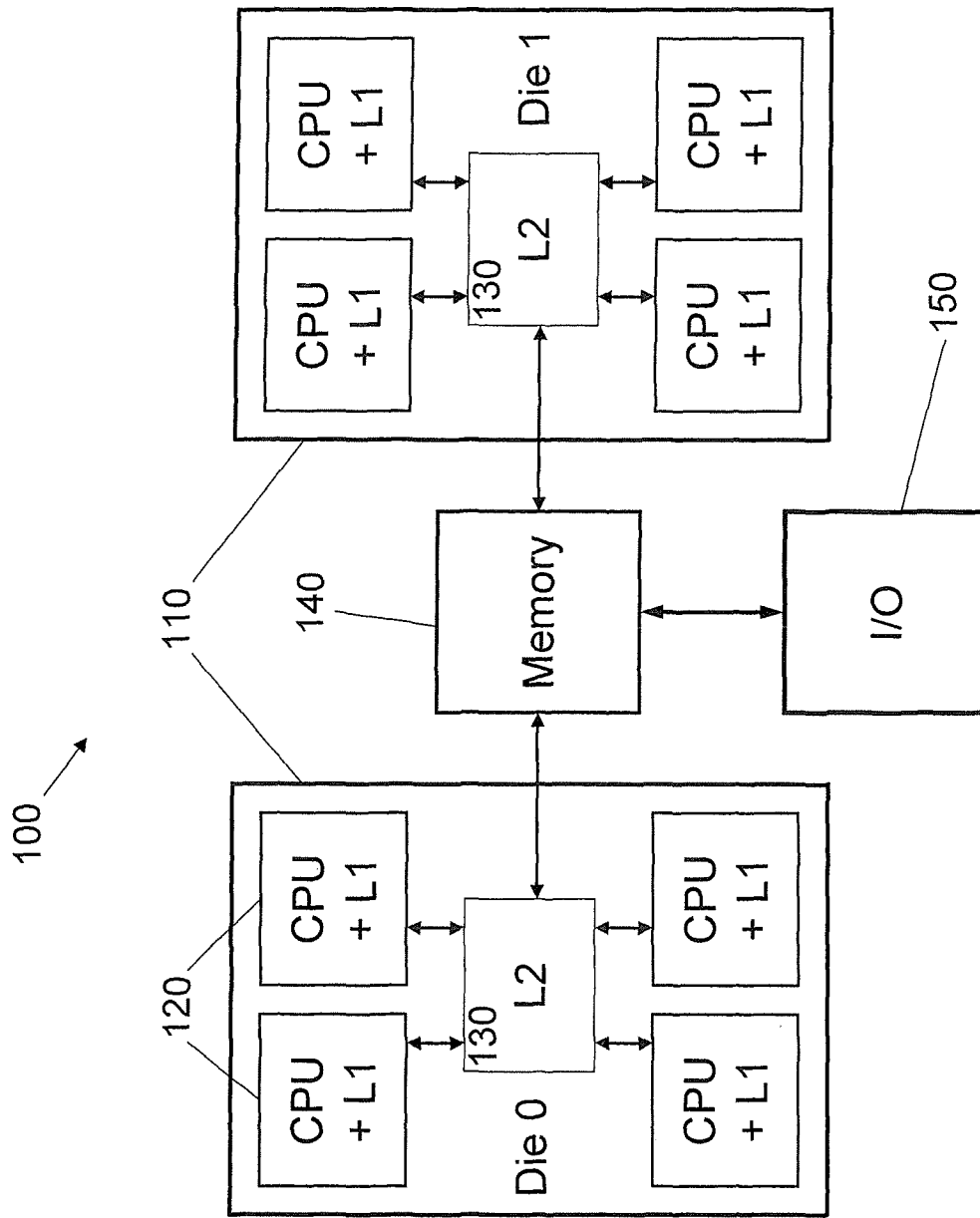
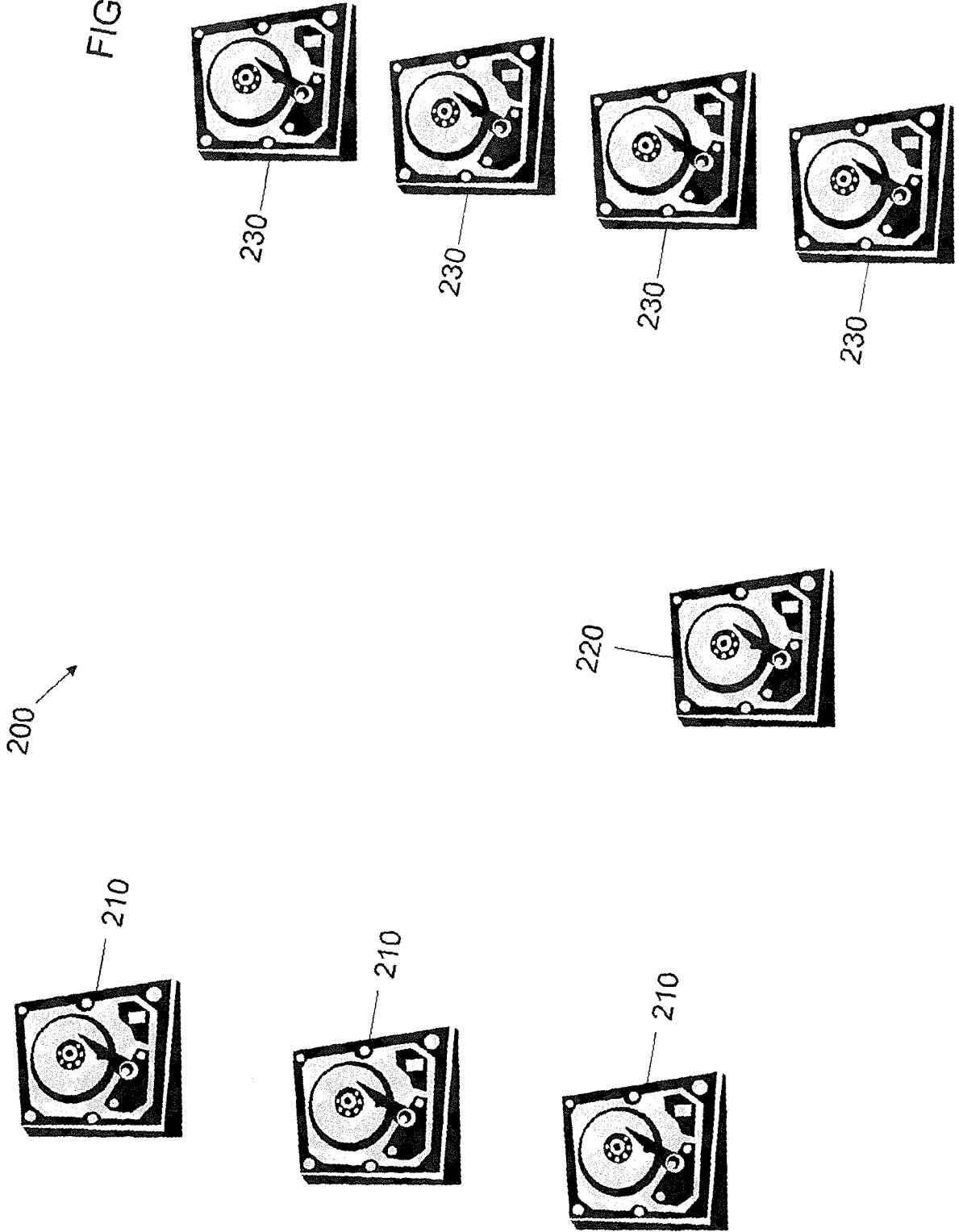


FIG. 9



US 8,683,296 B2

1

ACCELERATED ERASURE CODING SYSTEM AND METHOD

BACKGROUND

1. Field

Aspects of embodiments of the present invention are directed toward an accelerated erasure coding system and method.

2. Description of Related Art

An erasure code is a type of error-correcting code (ECC) useful for forward error-correction in applications like a redundant array of independent disks (RAID) or high-speed communication systems. In a typical erasure code, data (or original data) is organized in stripes, each of which is broken up into N equal-sized blocks, or data blocks, for some positive integer N . The data for each stripe is thus reconstructable by putting the N data blocks together. However, to handle situations where one or more of the original N data blocks gets lost, erasure codes also encode an additional M equal-sized blocks (called check blocks or check data) from the original N data blocks, for some positive integer M .

The N data blocks and the M check blocks are all the same size. Accordingly, there are a total of $N+M$ equal-sized blocks after encoding. The $N+M$ blocks may, for example, be transmitted to a receiver as $N+M$ separate packets, or written to $N+M$ corresponding disk drives. For ease of description, all $N+M$ blocks after encoding will be referred to as encoded blocks, though some (for example, N of them) may contain unencoded portions of the original data. That is, the encoded data refers to the original data together with the check data.

The M check blocks build redundancy into the system, in a very efficient manner, in that the original data (as well as any lost check data) can be reconstructed if any N of the $N+M$ encoded blocks are received by the receiver, or if any N of the $N+M$ disk drives are functioning correctly. Note that such an erasure code is also referred to as “optimal.” For ease of description, only optimal erasure codes will be discussed in this application. In such a code, up to M of the encoded blocks can be lost, (e.g., up to M of the disk drives can fail) so that if any N of the $N+M$ encoded blocks are received successfully by the receiver, the original data (as well as the check data) can be reconstructed. $N/(N+M)$ is thus the code rate of the erasure code encoding (i.e., how much space the original data takes up in the encoded data). Erasure codes for select values of N and M can be implemented on RAID systems employing $N+M$ (disk) drives by spreading the original data among N “data” drives, and using the remaining M drives as “check” drives. Then, when any N of the $N+M$ drives are correctly functioning, the original data can be reconstructed, and the check data can be regenerated.

Erasure codes (or more specifically, erasure coding systems) are generally regarded as impractical for values of M larger than 1 (e.g., RAID5 systems, such as parity drive systems) or 2 (RAID6 systems), that is, for more than one or two check drives. For example, see H. Peter Anvin, “The mathematics of RAID-6,” the entire content of which is incorporated herein by reference, p. 7, “Thus, in 2-disk-degraded mode, performance will be very slow. However, it is expected that that will be a rare occurrence, and that performance will not matter significantly in that case.” See also Robert Maddock et al., “Surviving Two Disk Failures,” p. 6, “The main difficulty with this technique is that calculating the check codes, and reconstructing data after failures, is quite complex. It involves polynomials and thus multiplication, and requires special hardware, or at least a signal processor, to do it at sufficient speed.” In addition, see also James S. Plank, “All

2

About Erasure Codes:—Reed-Solomon Coding—LDPC Coding,” slide 15 (describing computational complexity of Reed-Solomon decoding), “Bottom line: When n & m grow, it is brutally expensive.” Accordingly, there appears to be a general consensus among experts in the field that erasure coding systems are impractical for RAID systems for all but small values of M (that is, small numbers of check drives), such as 1 or 2.

Modern disk drives, on the other hand, are much less reliable than those envisioned when RAID was proposed. This is due to their capacity growing out of proportion to their reliability. Accordingly, systems with only a single check disk have, for the most part, been discontinued in favor of systems with two check disks.

In terms of reliability, a higher check disk count is clearly more desirable than a lower check disk count. If the count of error events on different drives is larger than the check disk count, data may be lost and that cannot be reconstructed from the correctly functioning drives. Error events extend well beyond the traditional measure of advertised mean time between failures (MTBF). A simple, real world example is a service event on a RAID system where the operator mistakenly replaces the wrong drive or, worse yet, replaces a good drive with a broken drive. In the absence of any generally accepted methodology to train, certify, and measure the effectiveness of service technicians, these types of events occur at an unknown rate, but certainly occur. The foolproof solution for protecting data in the face of multiple error events is to increase the check disk count.

SUMMARY

Aspects of embodiments of the present invention address these problems by providing a practical erasure coding system that, for byte-level RAID processing (where each byte is made up of 8 bits), performs well even for values of $N+M$ as large as 256 drives (for example, $N=127$ data drives and $M=129$ check drives). Further aspects provide for a single precomputed encoding matrix (or master encoding matrix) S of size $M_{max} \times N_{max}$ or $(N_{max}+M_{max}) \times N_{max}$ or $(M_{max}-1) \times N_{max}$ elements (e.g., bytes), which can be used, for example, for any combination of $N \leq N_{max}$ data drives and $M \leq M_{max}$ check drives such that $N_{max}+M_{max} \leq 256$ (e.g., $N_{max}=127$ and $M_{max}=129$, or $N_{max}=63$ and $M_{max}=193$). This is an improvement over prior art solutions that rebuild such matrices from scratch every time N or M changes (such as adding another check drive). Still higher values of N and M are possible with larger processing increments, such as 2 bytes, which affords up to $N+M=65,536$ drives (such as $N=32,767$ data drives and $M=32,769$ check drives).

Higher check disk count can offer increased reliability and decreased cost. The higher reliability comes from factors such as the ability to withstand more drive failures. The decreased cost arises from factors such as the ability to create larger groups of data drives. For example, systems with two checks disks are typically limited to group sizes of 10 or fewer drives for reliability reasons. With a higher check disk count, larger groups are available, which can lead to fewer overall components for the same unit of storage and hence, lower cost.

Additional aspects of embodiments of the present invention further address these problems by providing a standard parity drive as part of the encoding matrix. For instance, aspects provide for a parity drive for configurations with up to 127 data drives and up to 128 (non-parity) check drives, for a total of up to 256 total drives including the parity drive.

US 8,683,296 B2

3

Further aspects provide for different breakdowns, such as up to 63 data drives, a parity drive, and up to 192 (non-parity) check drives. Providing a parity drive offers performance comparable to RAID5 in comparable circumstances (such as single data drive failures) while also being able to tolerate significantly larger numbers of data drive failures by including additional (non-parity) check drives.

Further aspects are directed to a system and method for implementing a fast solution matrix algorithm for Reed-Solomon codes. While known solution matrix algorithms compute an $N \times N$ solution matrix (see, for example, J. S. Plank, "A tutorial on Reed-Solomon coding for fault-tolerance in RAID-like systems," *Software—Practice & Experience*, 27(9):995-1012, September 1997, and J. S. Plank and Y. Ding, "Note: Correction to the 1997 tutorial on Reed-Solomon coding," Technical Report CS-03-504, University of Tennessee, April 2003), requiring $O(N^3)$ operations, regardless of the number of failed data drives, aspects of embodiments of the present invention compute only an $F \times F$ solution matrix, where F is the number of failed data drives. The overhead for computing this $F \times F$ solution matrix is approximately $F^3/3$ multiplication operations and the same number of addition operations. Not only is $F \leq N$, in almost any practical application, the number of failed data drives F is considerably smaller than the number of data drives N . Accordingly, the fast solution matrix algorithm is considerably faster than any known approach for practical values of F and N .

Still further aspects are directed toward fast implementations of the check data generation and the lost (original and check) data reconstruction. Some of these aspects are directed toward fetching the surviving (original and check) data a minimum number of times (that is, at most once) to carry out the data reconstruction. Some of these aspects are directed toward efficient implementations that can maximize or significantly leverage the available parallel processing power of multiple cores working concurrently on the check data generation and the lost data reconstruction. Existing implementations do not attempt to accelerate these aspects of the data generation and thus fail to achieve a comparable level of performance.

In an exemplary embodiment of the present invention, a system for accelerated error-correcting code (ECC) processing is provided. The system includes a processing core for executing computer instructions and accessing data from a main memory; and a non-volatile storage medium (for example, a disk drive, or flash memory) for storing the computer instructions. The processing core, the storage medium, and the computer instructions are configured to implement an erasure coding system. The erasure coding system includes a data matrix for holding original data in the main memory, a check matrix for holding check data in the main memory, an encoding matrix for holding first factors in the main memory, and a thread for executing on the processing core. The first factors are for encoding the original data into the check data. The thread includes a parallel multiplier for concurrently multiplying multiple data entries of a matrix by a single factor; and a first sequencer for ordering operations through the data matrix and the encoding matrix using the parallel multiplier to generate the check data.

The first sequencer may be configured to access each entry of the data matrix from the main memory at most once while generating the check data.

The processing core may include a plurality of processing cores. The thread may include a plurality of threads. The erasure coding system may further include a scheduler for generating the check data by dividing the data matrix into a plurality of data matrices, dividing the check matrix into a

4

plurality of check matrices, assigning corresponding ones of the data matrices and the check matrices to the threads, and assigning the threads to the processing cores to concurrently generate portions of the check data corresponding to the check matrices from respective ones of the data matrices.

The data matrix may include a first number of rows. The check matrix may include a second number of rows. The encoding matrix may include the second number of rows and the first number of columns.

The data matrix may be configured to add rows to the first number of rows or the check matrix may be configured to add rows to the second number of rows while the first factors remain unchanged.

Each of entries of one of the rows of the encoding matrix may include a multiplicative identity factor (such as 1).

The data matrix may be configured to be divided by rows into a surviving data matrix for holding surviving original data of the original data, and a lost data matrix corresponding to lost original data of the original data and including a third number of rows. The erasure coding system may further include a solution matrix for holding second factors in the main memory. The second factors are for decoding the check data into the lost original data using the surviving original data and the first factors.

The solution matrix may include the third number of rows and the third number of columns.

The solution matrix may further include an inverted said third number by said third number sub-matrix of the encoding matrix.

The erasure coding system may further include a first list of rows of the data matrix corresponding to the surviving data matrix, and a second list of rows of the data matrix corresponding to the lost data matrix.

The data matrix may be configured to be divided into a surviving data matrix for holding surviving original data of the original data, and a lost data matrix corresponding to lost original data of the original data. The erasure coding system may further include a solution matrix for holding second factors in the main memory. The second factors are for decoding the check data into the lost original data using the surviving original data and the first factors. The thread may further include a second sequencer for ordering operations through the surviving data matrix, the encoding matrix, the check matrix, and the solution matrix using the parallel multiplier to reconstruct the lost original data.

The second sequencer may be further configured to access each entry of the surviving data matrix from the main memory at most once while reconstructing the lost original data.

The processing core may include a plurality of processing cores. The thread may include a plurality of threads. The erasure coding system may further include: a scheduler for generating the check data and reconstructing the lost original data by dividing the data matrix into a plurality of data matrices; dividing the surviving data matrix into a plurality of surviving data matrices; dividing the lost data matrix into a plurality of lost data matrices; dividing the check matrix into a plurality of check matrices; assigning corresponding ones of the data matrices, the surviving data matrices, the lost data matrices, and the check matrices to the threads; and assigning the threads to the processing cores to concurrently generate portions of the check data corresponding to the check matrices from respective ones of the data matrices and to concurrently reconstruct portions of the lost original data corresponding to the lost data matrices from respective ones of the surviving data matrices and the check matrices.

The check matrix may be configured to be divided into a surviving check matrix for holding surviving check data of

US 8,683,296 B2

5

the check data, and a lost check matrix corresponding to lost check data of the check data. The second sequencer may be configured to order operations through the surviving data matrix, the reconstructed lost original data, and the encoding matrix using the parallel multiplier to regenerate the lost check data.

The second sequencer may be further configured to reconstruct the lost original data concurrently with regenerating the lost check data.

The second sequencer may be further configured to access each entry of the surviving data matrix from the main memory at most once while reconstructing the lost original data and regenerating the lost check data.

The second sequencer may be further configured to regenerate the lost check data without accessing the reconstructed lost original data from the main memory.

The processing core may include a plurality of processing cores. The thread may include a plurality of threads. The erasure coding system may further include a scheduler for generating the check data, reconstructing the lost original data, and regenerating the lost check data by: dividing the data matrix into a plurality of data matrices; dividing the surviving data matrix into a plurality of surviving data matrices; dividing the lost data matrix into a plurality of lost data matrices; dividing the check matrix into a plurality of check matrices; dividing the surviving check matrix into a plurality of surviving check matrices; dividing the lost check matrix into a plurality of lost check matrices; assigning corresponding ones of the data matrices, the surviving data matrices, the lost data matrices, the check matrices, the surviving check matrices, and the lost check matrices to the threads; and assigning the threads to the processing cores to concurrently generate portions of the check data corresponding to the check matrices from respective ones of the data matrices, to concurrently reconstruct portions of the lost original data corresponding to the lost data matrices from respective ones of the surviving data matrices and the surviving check matrices, and to concurrently regenerate portions of the lost check data corresponding to the lost check matrices from respective ones of the surviving data matrices and respective portions of the reconstructed lost original data.

The processing core may include 16 data registers. Each of the data registers may include 16 bytes. The parallel multiplier may be configured to process the data in units of at least 64 bytes spread over at least four of the data registers at a time.

Consecutive instructions to process each of the units of the data may access separate ones of the data registers to permit concurrent execution of the consecutive instructions by the processing core.

The parallel multiplier may include two lookup tables for doing concurrent multiplication of 4-bit quantities across 16 byte-sized entries using the PSHUFB (Packed Shuffle Bytes) instruction.

The parallel multiplier may be further configured to receive an input operand in four of the data registers, and return with the input operand intact in the four of the data registers.

According to another exemplary embodiment of the present invention, a method of accelerated error-correcting code (ECC) processing on a computing system is provided. The computing system includes a non-volatile storage medium (such as a disk drive or flash memory), a processing core for accessing instructions and data from a main memory, and a computer program including a plurality of computer instructions for implementing an erasure coding system. The method includes: storing the computer program on the storage medium; executing the computer instructions on the processing core; arranging original data as a data matrix in the

6

main memory; arranging first factors as an encoding matrix in the main memory, the first factors being for encoding the original data into check data, the check data being arranged as a check matrix in the main memory; and generating the check data using a parallel multiplier for concurrently multiplying multiple data entries of a matrix by a single factor. The generating of the check data includes ordering operations through the data matrix and the encoding matrix using the parallel multiplier.

The generating of the check data may include accessing each entry of the data matrix from the main memory at most once.

The processing core may include a plurality of processing cores. The executing of the computer instructions may include executing the computer instructions on the processing cores. The method may further include scheduling the generating of the check data by: dividing the data matrix into a plurality of data matrices; dividing the check matrix into a plurality of check matrices; and assigning corresponding ones of the data matrices and the check matrices to the processing cores to concurrently generate portions of the check data corresponding to the check matrices from respective ones of the data matrices.

The method may further include: dividing the data matrix into a surviving data matrix for holding surviving original data of the original data, and a lost data matrix corresponding to lost original data of the original data; arranging second factors as a solution matrix in the main memory, the second factors being for decoding the check data into the lost original data using the surviving original data and the first factors; and reconstructing the lost original data by ordering operations through the surviving data matrix, the encoding matrix, the check matrix, and the solution matrix using the parallel multiplier.

The reconstructing of the lost original data may include accessing each entry of the surviving data matrix from the main memory at most once.

The processing core may include a plurality of processing cores. The executing of the computer instructions may include executing the computer instructions on the processing cores. The method may further include scheduling the generating of the check data and the reconstructing of the lost original data by: dividing the data matrix into a plurality of data matrices; dividing the surviving data matrix into a plurality of surviving data matrices; dividing the lost data matrix into a plurality of lost data matrices; dividing the check matrix into a plurality of check matrices; and assigning corresponding ones of the data matrices, the surviving data matrices, the lost data matrices, and the check matrices to the processing cores to concurrently generate portions of the check data corresponding to the check matrices from respective ones of the data matrices and to concurrently reconstruct portions of the lost original data corresponding to the lost data matrices from respective ones of the surviving data matrices and the check matrices.

The method may further include: dividing the check matrix into a surviving check matrix for holding surviving check data of the check data, and a lost check matrix corresponding to lost check data of the check data; and regenerating the lost check data by ordering operations through the surviving data matrix, the reconstructed lost original data, and the encoding matrix using the parallel multiplier.

The reconstructing of the lost original data may take place concurrently with the regenerating of the lost check data.

US 8,683,296 B2

7

The reconstructing of the lost original data and the regenerating of the lost check data may include accessing each entry of the surviving data matrix from the main memory at most once.

The regenerating of the lost check data may take place without accessing the reconstructed lost original data from the main memory.

The processing core may include a plurality of processing cores. The executing of the computer instructions may include executing the computer instructions on the processing cores. The method may further include scheduling the generating of the check data, the reconstructing of the lost original data, and the regenerating of the lost check data by: dividing the data matrix into a plurality of data matrices; dividing the surviving data matrix into a plurality of surviving data matrices; dividing the lost data matrix into a plurality of lost data matrices; dividing the check matrix into a plurality of check matrices; dividing the surviving check matrix into a plurality of surviving check matrices; dividing the lost check matrix into a plurality of lost check matrices; and assigning corresponding ones of the data matrices, the surviving data matrices, the lost data matrices, the check matrices, the surviving check matrices, and the lost check matrices to the processing cores to concurrently generate portions of the check data corresponding to the check matrices from respective ones of the data matrices, to concurrently reconstruct portions of the lost original data corresponding to the lost data matrices from respective ones of the surviving data matrices and the surviving check matrices, and to concurrently regenerate portions of the lost check data corresponding to the lost check matrices from respective ones of the surviving data matrices and respective portions of the reconstructed lost original data.

According to yet another exemplary embodiment of the present invention, a non-transitory computer-readable storage medium (such as a disk drive, a compact disk (CD), a digital video disk (DVD), flash memory, a universal serial bus (USB) drive, etc.) containing a computer program including a plurality of computer instructions for performing accelerated error-correcting code (ECC) processing on a computing system is provided. The computing system includes a processing core for accessing instructions and data from a main memory. The computer instructions are configured to implement an erasure coding system when executed on the computing system by performing the steps of: arranging original data as a data matrix in the main memory; arranging first factors as an encoding matrix in the main memory, the first factors being for encoding the original data into check data, the check data being arranged as a check matrix in the main memory; and generating the check data using a parallel multiplier for concurrently multiplying multiple data entries of a matrix by a single factor. The generating of the check data includes ordering operations through the data matrix and the encoding matrix using the parallel multiplier.

The generating of the check data may include accessing each entry of the data matrix from the main memory at most once.

The processing core may include a plurality of processing cores. The computer instructions may be further configured to perform the step of scheduling the generating of the check data by: dividing the data matrix into a plurality of data matrices; dividing the check matrix into a plurality of check matrices; and assigning corresponding ones of the data matrices and the check matrices to the processing cores to concurrently generate portions of the check data corresponding to the check matrices from respective ones of the data matrices.

8

The computer instructions may be further configured to perform the steps of: dividing the data matrix into a surviving data matrix for holding surviving original data of the original data, and a lost data matrix corresponding to lost original data of the original data; arranging second factors as a solution matrix in the main memory, the second factors being for decoding the check data into the lost original data using the surviving original data and the first factors; and reconstructing the lost original data by ordering operations through the surviving data matrix, the encoding matrix, the check matrix, and the solution matrix using the parallel multiplier.

The computer instructions may be further configured to perform the steps of: dividing the check matrix into a surviving check matrix for holding surviving check data of the check data, and a lost check matrix corresponding to lost check data of the check data; and regenerating the lost check data by ordering operations through the surviving data matrix, the reconstructed lost original data, and the encoding matrix using the parallel multiplier.

The reconstructing of the lost original data and the regenerating of the lost check data may include accessing each entry of the surviving data matrix from the main memory at most once.

The processing core may include a plurality of processing cores. The computer instructions may be further configured to perform the step of scheduling the generating of the check data, the reconstructing of the lost original data, and the regenerating of the lost check data by: dividing the data matrix into a plurality of data matrices; dividing the surviving data matrix into a plurality of surviving data matrices; dividing the lost data matrix into a plurality of lost data matrices; dividing the check matrix into a plurality of check matrices; dividing the surviving check matrix into a plurality of surviving check matrices; dividing the lost check matrix into a plurality of lost check matrices; and assigning corresponding ones of the data matrices, the surviving data matrices, the lost data matrices, the check matrices, the surviving check matrices, and the lost check matrices to the processing cores to concurrently generate portions of the check data corresponding to the check matrices from respective ones of the data matrices, to concurrently reconstruct portions of the lost original data corresponding to the lost data matrices from respective ones of the surviving data matrices and the surviving check matrices, and to concurrently regenerate portions of the lost check data corresponding to the lost check matrices from respective ones of the surviving data matrices and respective portions of the reconstructed lost original data.

By providing practical and efficient systems and methods for erasure coding systems (which for byte-level processing can support up to $N+M=256$ drives, such as $N=127$ data drives and $M=129$ check drives, including a parity drive), applications such as RAID systems that can tolerate far more failing drives than was thought to be possible or practical can be implemented with accelerated performance significantly better than any prior art solution.

BRIEF DESCRIPTION OF THE DRAWINGS

The accompanying drawings, together with the specification, illustrate exemplary embodiments of the present invention and, together with the description, serve to explain aspects and principles of the present invention.

FIG. 1 shows an exemplary stripe of original and check data according to an embodiment of the present invention.

FIG. 2 shows an exemplary method for reconstructing lost data after a failure of one or more drives according to an embodiment of the present invention.

FIG. 3 shows an exemplary method for performing a parallel lookup Galois field multiplication according to an embodiment of the present invention.

FIG. 4 shows an exemplary method for sequencing the parallel lookup multiplier to perform the check data generation according to an embodiment of the present invention.

FIGS. 5-7 show an exemplary method for sequencing the parallel lookup multiplier to perform the lost data reconstruction according to an embodiment of the present invention.

FIG. 8 illustrates a multi-core architecture system according to an embodiment of the present invention.

FIG. 9 shows an exemplary disk drive configuration according to an embodiment of the present invention.

DETAILED DESCRIPTION

Hereinafter, exemplary embodiments of the invention will be described in more detail with reference to the accompanying drawings. In the drawings, like reference numerals refer to like elements throughout.

While optimal erasure codes have many applications, for ease of description, they will be described in this application with respect to RAID applications, i.e., erasure coding systems for the storage and retrieval of digital data distributed across numerous storage devices (or drives), though the present application is not limited thereto. For further ease of description, the storage devices will be assumed to be disk drives, though the invention is not limited thereto. In RAID systems, the data (or original data) is broken up into stripes, each of which includes N uniformly sized blocks (data blocks), and the N blocks are written across N separate drives (the data drives), one block per data drive.

In addition, for ease of description, blocks will be assumed to be composed of L elements, each element having a fixed size, say 8 bits or one byte. An element, such as a byte, forms the fundamental unit of operation for the RAID processing, but the invention is just as applicable to other size elements, such as 16 bits (2 bytes). For simplification, unless otherwise indicated, elements will be assumed to be one byte in size throughout the description that follows, and the term “element(s)” and “byte(s)” will be used synonymously.

Conceptually, different stripes can distribute their data blocks across different combinations of drives, or have different block sizes or numbers of blocks, etc., but for simplification and ease of description and implementation, the described embodiments in the present application assume a consistent block size (L bytes) and distribution of blocks among the data drives between stripes. Further, all variables, such as the number of data drives N, will be assumed to be positive integers unless otherwise specified. In addition, since the N=1 case reduces to simple data mirroring (that is, copying the same data drive multiple times), it will also be assumed for simplicity that N≥2 throughout.

The N data blocks from each stripe are combined using arithmetic operations (to be described in more detail below) in M different ways to produce M blocks of check data (check blocks), and the M check blocks written across M drives (the check drives) separate from the N data drives, one block per check drive. These combinations can take place, for example, when new (or changed) data is written to (or back to) disk. Accordingly, each of the N+M drives (data drives and check drives) stores a similar amount of data, namely one block for each stripe. As the processing of multiple stripes is conceptually similar to the processing of one stripe (only processing multiple blocks per drive instead of one), it will be further assumed for simplification that the data being stored or retrieved is only one stripe in size unless otherwise indicated.

It will also be assumed that the block size L is sufficiently large that the data can be consistently divided across each block to produce subsets of the data that include respective portions of the blocks (for efficient concurrent processing by different processing units).

FIG. 1 shows an exemplary stripe 10 of original and check data according to an embodiment of the present invention.

Referring to FIG. 1, the stripe 10 can be thought of not only as the original N data blocks 20 that make up the original data, but also the corresponding M check blocks 30 generated from the original data (that is, the stripe 10 represents encoded data). Each of the N data blocks 20 is composed of L bytes 25 (labeled byte 1, byte 2, . . . , byte L), and each of the M check blocks 30 is composed of L bytes 35 (labeled similarly). In addition, check drive 1, byte 1, is a linear combination of data drive 1, byte 1; data drive 2, byte 1; . . . ; data drive N, byte 1. Likewise, check drive 1, byte 2, is generated from the same linear combination formula as check drive 1, byte 1, only using data drive 1, byte 2; data drive 2, byte 2; . . . ; data drive N, byte 2. In contrast, check drive 2, byte 1, uses a different linear combination formula than check drive 1, byte 1, but applies it to the same data, namely data drive 1, byte 1; data drive 2, byte 1; . . . ; data drive N, byte 1. In this fashion, each of the other check bytes 35 is a linear combination of the respective bytes of each of the N data drives 20 and using the corresponding linear combination formula for the particular check drive 30.

The stripe 10 in FIG. 1 can also be represented as a matrix C of encoded data. C has two sub-matrices, namely original data D on top and check data J on bottom. That is,

$$C = \begin{bmatrix} D \\ J \end{bmatrix} = \begin{bmatrix} D_{11} & D_{12} & \dots & D_{1L} \\ D_{21} & D_{22} & \dots & D_{2L} \\ \vdots & \vdots & \ddots & \vdots \\ D_{N1} & D_{N2} & \dots & D_{NL} \\ J_{11} & J_{12} & \dots & J_{1L} \\ J_{21} & J_{22} & \dots & J_{2L} \\ \vdots & \vdots & \ddots & \vdots \\ J_{M1} & J_{M2} & \dots & J_{ML} \end{bmatrix},$$

where D_{ij} =byte j from data drive i and J_{ij} =byte j from check drive i. Thus, the rows of encoded data C represent blocks, while the columns represent corresponding bytes of each of the drives.

Further, in case of a disk drive failure of one or more disks, the arithmetic operations are designed in such a fashion that for any stripe, the original data (and by extension, the check data) can be reconstructed from any combination of N data and check blocks from the corresponding N+M data and check blocks that comprise the stripe. Thus, RAID provides both parallel processing (reading and writing the data in stripes across multiple drives concurrently) and fault tolerance (regeneration of the original data even if as many as M of the drives fail), at the computational cost of generating the check data any time new data is written to disk, or changed data is written back to disk, as well as the computational cost of reconstructing any lost original data and regenerating any lost check data after a disk failure.

For example, for M=1 check drive, a single parity drive can function as the check drive (i.e., a RAID4 system). Here, the arithmetic operation is bitwise exclusive OR of each of the N corresponding data bytes in each data block of the stripe. In addition, as mentioned earlier, the assignment of parity blocks from different stripes to the same drive (i.e., RAID4)

US 8,683,296 B2

11

or different drives (i.e., RAID5) is arbitrary, but it does simplify the description and implementation to use a consistent assignment between stripes, so that will be assumed throughout. Since $M=1$ reduces to the case of a single parity drive, it will further be assumed for simplicity that $M \geq 2$ throughout.

For such larger values of M , Galois field arithmetic is used to manipulate the data, as described in more detail later. Galois field arithmetic, for Galois fields of powers-of-2 (such as 2^P) numbers of elements, includes two fundamental operations: (1) addition (which is just bitwise exclusive OR, as with the parity drive-only operations for $M=1$), and (2) multiplication. While Galois field (GF) addition is trivial on standard processors, GF multiplication is not. Accordingly, a significant component of RAID performance for $M \geq 2$ is speeding up the performance of GF multiplication, as will be discussed later. For purposes of description, GF addition will be represented by the symbol $+$ throughout while GF multiplication will be represented by the symbol \times throughout.

Briefly, in exemplary embodiments of the present invention, each of the M check drives holds linear combinations (over GF arithmetic) of the N data drives of original data, one linear combination (i.e., a GF sum of N terms, where each term represents a byte of original data times a corresponding factor (using GF multiplication) for the respective data drive) for each check drive, as applied to respective bytes in each block. One such linear combination can be a simple parity, i.e., entirely GF addition (all factors equal 1), such as a GF sum of the first byte in each block of original data as described above.

The remaining $M-1$ linear combinations include more involved calculations that include the nontrivial GF multiplication operations (e.g., performing a GF multiplication of the first byte in each block by a corresponding factor for the respective data drive, and then performing a GF sum of all these products). These linear combinations can be represented by an $(N+M) \times N$ matrix (encoding matrix or information dispersal matrix (IDM)) E of the different factors, one factor for each combination of (data or check) drive and data drive, with one row for each of the $N+M$ data and check drives and one column for each of the N data drives. The IDM E can also be represented as

$$\begin{bmatrix} I_N \\ H \end{bmatrix},$$

where I_N represents the $N \times N$ identity matrix (i.e., the original (unencoded) data) and H represents the $M \times N$ matrix of factors for the check drives (where each of the M rows corresponds to one of the M check drives and each of the N columns corresponds to one of the N data drives).

Thus,

$$E = \begin{bmatrix} I_N \\ H \end{bmatrix} = \begin{bmatrix} 1 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 1 \\ H_{11} & H_{12} & \dots & H_{1N} \\ H_{21} & H_{22} & \dots & H_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ H_{M1} & H_{M2} & \dots & H_{MN} \end{bmatrix},$$

12

where H_{ij} =factor for check drive i and data drive j . Thus, the rows of encoded data C represent blocks, while the columns represent corresponding bytes of each of the drives. In addition, check factors H , original data D , and check data J are related by the formula $J=H \times D$ (that is, matrix multiplication), or

$$\begin{bmatrix} J_{11} & J_{12} & \dots & J_{1L} \\ J_{21} & J_{22} & \dots & J_{2L} \\ \vdots & \vdots & \ddots & \vdots \\ J_{M1} & J_{M2} & \dots & J_{ML} \end{bmatrix} = \begin{bmatrix} H_{11} & H_{12} & \dots & H_{1N} \\ H_{21} & H_{22} & \dots & H_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ H_{M1} & H_{M2} & \dots & H_{MN} \end{bmatrix} \times \begin{bmatrix} D_{11} & D_{12} & \dots & D_{1L} \\ D_{21} & D_{22} & \dots & D_{2L} \\ \vdots & \vdots & \ddots & \vdots \\ D_{N1} & D_{N2} & \dots & D_{NL} \end{bmatrix},$$

where $J_{11}=(H_{11} \times D_{11})+(H_{12} \times D_{21})+\dots+(H_{1N} \times D_{N1})$, $J_{12}=(H_{11} \times D_{12})+(H_{12} \times D_{22})+\dots+(H_{1N} \times D_{N2})$, $J_{21}=(H_{21} \times D_{11})+(H_{22} \times D_{21})+\dots+(H_{2N} \times D_{N1})$, and in general, $J_{ij}=(H_{i1} \times D_{1j})+(H_{i2} \times D_{2j})+\dots+(H_{iN} \times D_{Nj})$ for $1 \leq i \leq M$ and $1 \leq j \leq L$.

Such an encoding matrix E is also referred to as an information dispersal matrix (IDM). It should be noted that matrices such as check drive encoding matrix H and identity matrix I_N also represent encoding matrices, in that they represent matrices of factors to produce linear combinations over GF arithmetic of the original data. In practice, the identity matrix I_N is trivial and may not need to be constructed as part of the IDM E . Only the encoding matrix E , however, will be referred to as the IDM. Methods of building an encoding matrix such as IDM E or check drive encoding matrix H are discussed below. In further embodiments of the present invention (as discussed further in Appendix A), such $(N+M) \times N$ (or $M \times N$) matrices can be trivially constructed (or simply indexed) from a master encoding matrix S , which is composed of $(N_{max}+M_{max}) \times N_{max}$ (or $M_{max} \times N_{max}$) bytes or elements, where $N_{max}+M_{max}=256$ (or some other power of two) and $N \leq N_{max}$ and $M \leq M_{max}$. For example, one such master encoding matrix S can include a 127×127 element identity matrix on top (for up to $N_{max}=127$ data drives), a row of 1's (for a parity drive), and a 128×127 element encoding matrix on bottom (for up to $M_{max}=129$ check drives, including the parity drive), for a total of $N_{max}+M_{max}=256$ drives.

The original data, in turn, can be represented by an $N \times L$ matrix D of bytes, each of the N rows representing the L bytes of a block of the corresponding one of the N data drives. If C represents the corresponding $(N+M) \times L$ matrix of encoded bytes (where each of the $N+M$ rows corresponds to one of the $N+M$ data and check drives), then C can be represented as

$$E \times D = \begin{bmatrix} I_N \\ H \end{bmatrix} \times D = \begin{bmatrix} I_N \times D \\ H \times D \end{bmatrix} = \begin{bmatrix} D \\ J \end{bmatrix},$$

where $J=H \times D$ is an $M \times L$ matrix of check data, with each of the M rows representing the L check bytes of the corresponding one of the M check drives. It should be noted that in the relationships such as $C=E \times D$ or $J=H \times D$, \times represents matrix multiplication over the Galois field (i.e., GF multiplication and GF addition being used to generate each of the entries in, for example, C or J).

13

In exemplary embodiments of the present invention, the first row of the check drive encoding matrix H (or the (N+1)th row of the IDM E) can be all 1's representing the parity drive. For linear combinations involving this row, the GF multiplication can be bypassed and replaced with a GF sum of the corresponding bytes since the products are all trivial products involving the identity element 1. Accordingly, in parity drive implementations, the check drive encoding matrix H can also be thought of as an (M-1)×N matrix of non-trivial factors (that is, factors intended to be used in GF multiplication and not just GF addition).

Much of the RAID processing involves generating the check data when new or changed data is written to (or back to) disk. The other significant event for RAID processing is when one or more of the drives fail (data or check drives), or for whatever reason become unavailable. Assume that in such a failure scenario, F data drives fail and G check drives fail, where F and G are nonnegative integers. If F=0, then only check drives failed and all of the original data D survived. In this case, the lost check data can be regenerated from the original data D.

Accordingly, assume at least one data drive fails, that is, F≥1, and let K=N-F represent the number of data drives that survive. K is also a nonnegative integer. In addition, let X represent the surviving original data and Y represent the lost original data. That is, X is a K×L matrix composed of the K rows of the original data matrix D corresponding to the K surviving data drives, while Y is an F×L matrix composed of the F rows of the original data matrix D corresponding to the F failed data drives.

$$\begin{bmatrix} X \\ Y \end{bmatrix}$$

thus represents a permuted original data matrix D' (that is, the original data matrix D, only with the surviving original data X on top and the lost original data Y on bottom. It should be noted that once the lost original data Y is reconstructed, it can be combined with the surviving original data X to restore the original data D, from which the check data for any of the failed check drives can be regenerated.

It should also be noted that M-G check drives survive. In order to reconstruct the lost original data Y, enough (that is, at least N) total drives must survive. Given that K=N-F data drives survive, and that M-G check drives survive, it follows that (N-F)+(M-G)≥N must be true to reconstruct the lost original data Y. This is equivalent to F+G≤M (i.e., no more than F+G drives fail), or F≤M-G (that is, the number of failed data drives does not exceed the number of surviving check drives). It will therefore be assumed for simplicity that F≤M-G.

In the routines that follow, performance can be enhanced by prebuilding lists of the failed and surviving data and check drives (that is, four separate lists). This allows processing of the different sets of surviving and failed drives to be done more efficiently than existing solutions, which use, for example, bit vectors that have to be examined one bit at a time and often include large numbers of consecutive zeros (or ones) when ones (or zeros) are the bit values of interest.

FIG. 2 shows an exemplary method 300 for reconstructing lost data after a failure of one or more drives according to an embodiment of the present invention.

While the recovery process is described in more detail later, briefly it consists of two parts: (1) determining the solution matrix, and (2) reconstructing the lost data from the

14

surviving data. Determining the solution matrix can be done in three steps with the following algorithm (Algorithm 1), with reference to FIG. 2:

1. (Step 310 in FIG. 2) Reducing the (M+N)×N IDM E to an N×N reduced encoding matrix T (also referred to as the transformed IDM) including the K surviving data drive rows and any F of the M-G surviving check drive rows (for instance, the first F surviving check drive rows, as these will include the parity drive if it survived; recall that F≤M-G was assumed). In addition, the columns of the reduced encoding matrix T are rearranged so that the K columns corresponding to the K surviving data drives are on the left side of the matrix and the F columns corresponding to the F failed drives are on the right side of the matrix. (Step 320) These F surviving check drives selected to rebuild the lost original data Y will henceforth be referred to as "the F surviving check drives," and their check data W will be referred to as "the surviving check data," even though M-G check drives survived. It should be noted that W is an F×L matrix composed of the F rows of the check data J corresponding to the F surviving check drives. Further, the surviving encoded data can be represented as a sub-matrix C' of the encoded data C. The surviving encoded data C' is an N×L matrix composed of the surviving original data X on top and the surviving check data W on bottom, that is,

$$C' = \begin{bmatrix} X \\ W \end{bmatrix}.$$

2. (Step 330) Splitting the reduced encoding matrix T into four sub-matrices (that are also encoding matrices): (i) a K×K identity matrix I_K (corresponding to the K surviving data drives) in the upper left, (ii) a K×F matrix O of zeros in the upper right, (iii) an F×K encoding matrix A in the lower left corresponding to the F surviving check drive rows and the K surviving data drive columns, and (iv) an F×F encoding matrix B in the lower right corresponding to the F surviving check drive rows and the F failed data drive columns. Thus, the reduced encoding matrix T can be represented as

$$\begin{bmatrix} I_K & O \\ A & B \end{bmatrix}.$$

3. (Step 340) Calculating the inverse B of the F×F encoding matrix B. As is shown in more detail in Appendix A, C'=T×D, or

$$\begin{bmatrix} X \\ W \end{bmatrix} = \begin{bmatrix} I_K & O \\ A & B \end{bmatrix} \times \begin{bmatrix} X \\ Y \end{bmatrix},$$

which is mathematically equivalent to W=A×X+B×Y. B⁻¹ is the solution matrix, and is itself an F×F encoding matrix. Calculating the solution matrix B⁻¹ thus allows the lost original data Y to be reconstructed from the encoding matrices A and B along with the surviving original data X and the surviving check data W.

The F×K encoding matrix A represents the original encoding matrix E, only limited to the K surviving data drives and the F surviving check drives. That is, each of the F rows of A

US 8,683,296 B2

15

represents a different one of the F surviving check drives, while each of the K columns of A represents a different one of the K surviving data drives. Thus, A provides the encoding factors needed to encode the original data for the surviving check drives, but only applied to the surviving data drives (that is, the surviving partial check data). Since the surviving original data X is available, A can be used to generate this surviving partial check data.

In similar fashion, the $F \times F$ encoding matrix B represents the original encoding matrix E , only limited to the F surviving check drives and the F failed data drives. That is, the F rows of B correspond to the same F rows of A , while each of the F columns of B represents a different one of the F failed data drives. Thus, B provides the encoding factors needed to encode the original data for the surviving check drives, but only applied to the failed data drives (that is, the lost partial check data). Since the lost original data Y is not available, B cannot be used to generate any of the lost partial check data. However, this lost partial check data can be determined from A and the surviving check data W . Since this lost partial check data represents the result of applying B to the lost original data Y , B^{-1} thus represents the necessary factors to reconstruct the lost original data Y from the lost partial check data.

It should be noted that steps 1 and 2 in Algorithm 1 above are logical, in that encoding matrices A and B (or the reduced encoding matrix T , for that matter) do not have to actually be constructed. Appropriate indexing of the IDM E (or the master encoding matrix S) can be used to obtain any of their entries. Step 3, however, is a matrix inversion over GF arithmetic and takes $O(F^3)$ operations, as discussed in more detail later. Nonetheless, this is a significant improvement over existing solutions, which require $O(N^3)$ operations, since the number of failed data drives F is usually significantly less than the number of data drives N in any practical situation.

(Step 350 in FIG. 2) Once the encoding matrix A and the solution matrix B^{-1} are known, reconstructing the lost data from the surviving data (that is, the surviving original data X and the surviving check data W) can be accomplished in four steps using the following algorithm (Algorithm 2):

1. Use A and the surviving original data X (using matrix multiplication) to generate the surviving check data (i.e., $A \times X$), only limited to the K surviving data drives. Call this limited check data the surviving partial check data.
2. Subtract this surviving partial check data from the surviving check data W (using matrix subtraction, i.e., $W - A \times X$, which is just entry-by-entry GF subtraction, which is the same as GF addition for this Galois field). This generates the surviving check data, only this time limited to the F failed data drives. Call this limited check data the lost partial check data.
3. Use the solution matrix B^{-1} and the lost partial check data (using matrix multiplication, i.e., $B^{-1} \times (W - A \times X)$) to reconstruct the lost original data Y . Call this the recovered original data Y .
4. Use the corresponding rows of the IDM E (or master encoding matrix S) for each of the G failed check drives along with the original data D , as reconstructed from the surviving and recovered original data X and Y , to regenerate the lost check data (using matrix multiplication).

As will be shown in more detail later, steps 1-3 together require $O(F)$ operations times the amount of original data D to reconstruct the lost original data Y for the F failed data drives (i.e., roughly 1 operation per failed data drive per byte of original data D), which is proportionally equivalent to the $O(M)$ operations times the amount of original data D needed to generate the check data J for the M check drives (i.e., roughly 1 operation per check drive per byte of original data

16

D). In addition, this same equivalence extends to step 4, which takes $O(G)$ operations times the amount of original data D needed to regenerate the lost check data for the G failed check drives (i.e., roughly 1 operation per failed check drive per byte of original data D). In summary, the number of operations needed to reconstruct the lost data is $O(F+G)$ times the amount of original data D (i.e., roughly 1 operation per failed drive (data or check) per byte of original data D). Since $F+G \leq M$, this means that the computational complexity of Algorithm 2 (reconstructing the lost data from the surviving data) is no more than that of generating the check data J from the original data D .

As mentioned above, for exemplary purposes and ease of description, data is assumed to be organized in 8-bit bytes, each byte capable of taking on $2^8=256$ possible values. Such data can be manipulated in byte-size elements using GF arithmetic for a Galois field of size $2^8=256$ elements. It should also be noted that the same mathematical principles apply to any power-of-two 2^P number of elements, not just 256, as Galois fields can be constructed for any integral power of a prime number. Since Galois fields are finite, and since GF operations never overflow, all results are the same size as the inputs, for example, 8 bits.

In a Galois field of a power-of-two number of elements, addition and subtraction are the same operation, namely a bitwise exclusive OR (XOR) of the two operands. This is a very fast operation to perform on any current processor. It can also be performed on multiple bytes concurrently. Since the addition and subtraction operations take place, for example, on a byte-level basis, they can be done in parallel by using, for instance, x86 architecture Streaming SIMD Extensions (SSE) instructions (SIMD stands for single instruction, multiple data, and refers to performing the same instruction on different pieces of data, possibly concurrently), such as PXOR (Packed (bitwise) Exclusive OR).

SSE instructions can process, for example, 16-byte registers (XMM registers), and are able to process such registers as though they contain 16 separate one-byte operands (or 8 separate two-byte operands, or four separate four-byte operands, etc.) Accordingly, SSE instructions can do byte-level processing 16 times faster than when compared to processing a byte at a time. Further, there are 16 XMM registers, so dedicating four such registers for operand storage allows the data to be processed in 64-byte increments, using the other 12 registers for temporary storage. That is, individual operations can be performed as four consecutive SSE operations on the four respective registers (64 bytes), which can often allow such instructions to be efficiently pipelined and/or concurrently executed by the processor. In addition, the SSE instructions allows the same processing to be performed on different such 64-byte increments of data in parallel using different cores. Thus, using four separate cores can potentially speed up this processing by an additional factor of 4 over using a single core.

For example, a parallel adder (Parallel Adder) can be built using the 16-byte XMM registers and four consecutive PXOR instructions. Such parallel processing (that is, 64 bytes at a time with only a few machine-level instructions) for GF arithmetic is a significant improvement over doing the addition one byte at a time. Since the data is organized in blocks of any fixed number of bytes, such as 4096 bytes (4 kilobytes, or 4 KB) or 32,768 bytes (32 KB), a block can be composed of numerous such 64-byte chunks (e.g., 64 separate 64-byte chunks in 4 KB, or 512 chunks in 32 KB).

Multiplication in a Galois field is not as straightforward. While much of it is bitwise shifts and exclusive OR's (i.e., "additions") that are very fast operations, the numbers "wrap"

in peculiar ways when they are shifted outside of their normal bounds (because the field has only a finite set of elements), which can slow down the calculations. This “wrapping” in the GF multiplication can be addressed in many ways. For example, the multiplication can be implemented serially (Serial Multiplier) as a loop iterating over the bits of one operand while performing the shifts, adds, and wraps on the other operand. Such processing, however, takes several machine instructions per bit for 8 separate bits. In other words, this technique requires dozens of machine instructions per byte being multiplied. This is inefficient compared to, for example, the performance of the Parallel Adder described above.

For another approach (Serial Lookup Multiplier), multiplication tables (of all the possible products, or at least all the non-trivial products) can be pre-computed and built ahead of time. For example, a table of $256 \times 256 = 65,536$ bytes can hold all the possible products of the two different one-byte operands). However, such tables can force serialized access on what are only byte-level operations, and not take advantage of wide (concurrent) data paths available on modern processors, such as those used to implement the Parallel Adder above.

In still another approach (Parallel Multiplier), the GF multiplication can be done on multiple bytes at a time, since the same factor in the encoding matrix is multiplied with every element in a data block. Thus, the same factor can be multiplied with 64 consecutive data block bytes at a time. This is similar to the Parallel Adder described above, only there are several more operations needed to perform the operation. While this can be implemented as a loop on each bit of the factor, as described above, only performing the shifts, adds, and wraps on 64 bytes at a time, it can be more efficient to process the 256 possible factors as a (C language) switch statement, with inline code for each of 256 different combinations of two primitive GF operations: Multiply-by-2 and Add. For example, GF multiplication by the factor 3 can be effected by first doing a Multiply-by-2 followed by an Add. Likewise, GF multiplication by 4 is just a Multiply-by-2 followed by a Multiply-by-2 while multiplication by 6 is a Multiply-by-2 followed by an Add and then by another Multiply-by-2.

While this Add is identical to the Parallel Adder described above (e.g., four consecutive PXOR instructions to process 64 separate bytes), Multiply-by-2 is not as straightforward. For example, Multiply-by-2 in GF arithmetic can be implemented across 64 bytes at a time in 4 XMM registers via 4 consecutive PXOR instructions, 4 consecutive PCMPGTB (Packed Compare for Greater Than) instructions, 4 consecutive PADDB (Packed Add) instructions, 4 consecutive PAND (Bitwise AND) instructions, and 4 consecutive PXOR instructions. Though this takes 20 machine instructions, the instructions are very fast and results in 64 consecutive bytes of data at a time being multiplied by 2.

For 64 bytes of data, assuming a random factor between 0 and 255, the total overhead for the Parallel Multiplier is about 6 calls to multiply-by-2 and about 3.5 calls to add, or about $6 \times 20 + 3.5 \times 4 = 134$ machine instructions, or a little over 2 machine instructions per byte of data. While this compares favorably with byte-level processing, it is still possible to improve on this by building a parallel multiplier with a table lookup (Parallel Lookup Multiplier) using the PSHUFB (Packed Shuffle Bytes) instruction and doing the GF multiplication in 4-bit nibbles (half bytes).

FIG. 3 shows an exemplary method 400 for performing a parallel lookup Galois field multiplication according to an embodiment of the present invention.

Referring to FIG. 3, in step 410, two lookup tables are built once: one lookup table for the low-order nibbles in each byte, and one lookup table for the high-order nibbles in each byte.

Each lookup table contains 256 sets (one for each possible factor) of the 16 possible GF products of that factor and the 16 possible nibble values. Each lookup table is thus $256 \times 16 = 4096$ bytes, which is considerably smaller than the 65,536 bytes needed to store a complete one-byte multiplication table. In addition, PSHUFB does 16 separate table lookups at once, each for one nibble, so 8 PSHUFB instructions can be used to do all the table lookups for 64 bytes (128 nibbles).

Next, in step 420, the Parallel Lookup Multiplier is initialized for the next set of 64 bytes of operand data (such as original data or surviving original data). In order to save loading this data from memory on succeeding calls, the Parallel Lookup Multiplier dedicates four registers for this data, which are left intact upon exit of the Parallel Lookup Multiplier. This allows the same data to be called with different factors (such as processing the same data for another check drive).

Next in step 430, to process these 64 bytes of operand data, the Parallel Lookup Multiplier can be implemented with 2 MOVDQA (Move Double Quadword Aligned) instructions (from memory) to do the two table lookups and 4 MOVDQA instructions (register to register) to initialize registers (such as the output registers). These are followed in steps 440 and 450 by two nearly identical sets of 17 register-to-register instructions to carry out the multiplication 32 bytes at a time. Each such set starts (in step 440) with 5 more MOVDQA instructions for further initialization, followed by 2 PSRLW (Packed Shift Right Logical Word) instructions to realign the high-order nibbles for PSHUFB, and 4 PAND instructions to clear the high-order nibbles for PSHUFB. That is, two registers of byte operands are converted into four registers of nibble operands. Then, in step 450, 4 PSHUFB instructions are used to do the parallel table lookups, and 2 PXOR instructions to add the results of the multiplication on the two nibbles to the output registers.

Thus, the Parallel Lookup Multiplier uses 40 machine instructions to perform the parallel multiplication on 64 separate bytes, which is considerably better than the average 134 instructions for the Parallel Multiplier above, and only 10 times as many instructions as needed for the Parallel Adder. While some of the Parallel Lookup Multiplier’s instructions are more complex than those of the Parallel Adder, much of this complexity can be concealed through the pipelined and/or concurrent execution of numerous such contiguous instructions (accessing different registers) on modern pipelined processors. For example, in exemplary implementations, the Parallel Lookup Multiplier has been timed at about 15 CPU clock cycles per 64 bytes processed per CPU core (about 0.36 clock cycles per instruction). In addition, the code footprint is practically nonexistent for the Parallel Lookup Multiplier (40 instructions) compared to that of the Parallel Multiplier (about 34,300 instructions), even when factoring the 8 KB needed for the two lookup tables in the Parallel Lookup Multiplier.

In addition, embodiments of the Parallel Lookup Multiplier can be passed 64 bytes of operand data (such as the next 64 bytes of surviving original data X to be processed) in four consecutive registers, whose contents can be preserved upon exiting the Parallel Lookup Multiplier (and all in the same 40 machine instructions) such that the Parallel Lookup Multiplier can be invoked again on the same 64 bytes of data without having to access main memory to reload the data. Through such a protocol, memory accesses can be minimized (or significantly reduced) for accessing the original data D

during check data generation or the surviving original data X during lost data reconstruction.

Further embodiments of the present invention are directed towards sequencing this parallel multiplication (and other GF) operations. While the Parallel Lookup Multiplier processes a GF multiplication of 64 bytes of contiguous data times a specified factor, the calls to the Parallel Lookup Multiplier should be appropriately sequenced to provide efficient processing. One such sequencer (Sequencer 1), for example, can generate the check data J from the original data D, and is described further with respect to FIG. 4.

The parity drive does not need GF multiplication. The check data for the parity drive can be obtained, for example, by adding corresponding 64-byte chunks for each of the data drives to perform the parity operation. The Parallel Adder can do this using 4 instructions for every 64 bytes of data for each of the N data drives, or N/16 instructions per byte.

The M-1 non-parity check drives can invoke the Parallel Lookup Multiplier on each 64-byte chunk, using the appropriate factor for the particular combination of data drive and check drive. One consideration is how to handle the data access. Two possible ways are:

- 1) "column-by-column," i.e., 64 bytes for one data drive, followed by the next 64 bytes for that data drive, etc., and adding the products to the running total in memory (using the Parallel Adder) before moving onto the next row (data drive); and
- 2) "row-by-row," i.e., 64 bytes for one data drive, followed by the corresponding 64 bytes for the next data drive, etc., and keeping a running total using the Parallel Adder, then moving onto the next set of 64-byte chunks.

Column-by-column can be thought of as "constant factor, varying data," in that the (GF multiplication) factor usually remains the same between iterations while the (64-byte) data changes with each iteration. Conversely, row-by-row can be thought of as "constant data, varying factor," in that the data usually remains the same between iterations while the factor changes with each iteration.

Another consideration is how to handle the check drives. Two possible ways are:

- a) one at a time, i.e., generate all the check data for one check drive before moving onto the next check drive; and
- b) all at once, i.e., for each 64-byte chunk of original data, do all of the processing for each of the check drives before moving onto the next chunk of original data.

While each of these techniques performs the same basic operations (e.g., 40 instructions for every 64 bytes of data for each of the N data drives and M-1 non-parity check drives, or $5N(M-1)/8$ instructions per byte for the Parallel Lookup Multiplier), empirical results show that combination (2)(b), that is, row-by-row data access on all of the check drives between data accesses performs best with the Parallel Lookup Multiplier. One reason may be that such an approach appears to minimize the number of memory accesses (namely, one) to each chunk of the original data D to generate the check data J. This embodiment of Sequencer 1 is described in more detail with reference to FIG. 4.

FIG. 4 shows an exemplary method 500 for sequencing the Parallel Lookup Multiplier to perform the check data generation according to an embodiment of the present invention.

Referring to FIG. 4, in step 510, the Sequencer 1 is called. Sequencer 1 is called to process multiple 64-byte chunks of data for each of the blocks across a stripe of data. For instance, Sequencer 1 could be called to process 512 bytes from each block. If, for example, the block size L is 4096 bytes, then it would take eight such calls to Sequencer 1 to process the

entire stripe. The other such seven calls to Sequencer 1 could be to different processing cores, for instance, to carry out the check data generation in parallel. The number of 64-byte chunks to process at a time could depend on factors such as cache dimensions, input/output data structure sizes, etc.

In step 520, the outer loop processes the next 64-byte chunk of data for each of the drives. In order to minimize the number of accesses of each data drive's 64-byte chunk of data from memory, the data is loaded only once and preserved across calls to the Parallel Lookup Multiplier. The first data drive is handled specially since the check data has to be initialized for each check drive. Using the first data drive to initialize the check data saves doing the initialization as a separate step followed by updating it with the first data drive's data. In addition to the first data drive, the first check drive is also handled specially since it is a parity drive, so its check data can be initialized to the first data drive's data directly without needing the Parallel Lookup Multiplier.

In step 530, the first middle loop is called, in which the remainder of the check drives (that is, the non-parity check drives) have their check data initialized by the first data drive's data. In this case, there is a corresponding factor (that varies with each check drive) that needs to be multiplied with each of the first data drive's data bytes. This is handled by calling the Parallel Lookup Multiplier for each non-parity check drive.

In step 540, the second middle loop is called, which processes the other data drives' corresponding 64-byte chunks of data. As with the first data drive, each of the other data drives is processed separately, loading the respective 64 bytes of data into four registers (preserved across calls to the Parallel Lookup Multiplier). In addition, since the first check drive is the parity drive, its check data can be updated by directly adding these 64 bytes to it (using the Parallel Adder) before handling the non-parity check drives.

In step 550, the inner loop is called for the next data drive. In the inner loop (as with the first middle loop), each of the non-parity check drives is associated with a corresponding factor for the particular data drive. The factor is multiplied with each of the next data drive's data bytes using the Parallel Lookup Multiplier, and the results added to the check drive's check data.

Another such sequencer (Sequencer 2) can be used to reconstruct the lost data from the surviving data (using Algorithm 2). While the same column-by-column and row-by-row data access approaches are possible, as well as the same choices for handling the check drives, Algorithm 2 adds another dimension of complexity because of the four separate steps and whether to: (i) do the steps completely serially or (ii) do some of the steps concurrently on the same data. For example, step 1 (surviving check data generation) and step 4 (lost check data regeneration) can be done concurrently on the same data to reduce or minimize the number of surviving original data accesses from memory.

Empirical results show that method (2)(b)(ii), that is, row-by-row data access on all of the check drives and for both surviving check data generation and lost check data regeneration between data accesses performs best with the Parallel Lookup Multiplier when reconstructing lost data using Algorithm 2. Again, this may be due to the apparent minimization of the number of memory accesses (namely, one) of each chunk of surviving original data X to reconstruct the lost data and the absence of memory accesses of reconstructed lost original data Y when regenerating the lost check data. This embodiment of Sequencer 1 is described in more detail with reference to FIGS. 5-7.

US 8,683,296 B2

21

FIGS. 5-7 show an exemplary method 600 for sequencing the Parallel Lookup Multiplier to perform the lost data reconstruction according to an embodiment of the present invention.

Referring to FIG. 5, in step 610, the Sequencer 2 is called. Sequencer 2 has many similarities with the embodiment of Sequencer 1 illustrated in FIG. 4. For instance, Sequencer 2 processes the data drive data in 64-byte chunks like Sequencer 1. Sequencer 2 is more complex, however, in that only some of the data drive data is surviving; the rest has to be reconstructed. In addition, lost check data needs to be regenerated. Like Sequencer 1, Sequencer 2 does these operations in such a way as to minimize memory accesses of the data drive data (by loading the data once and calling the Parallel Lookup Multiplier multiple times). Assume for ease of description that there is at least one surviving data drive; the case of no surviving data drives is handled a little differently, but not significantly different. In addition, recall from above that the driving formula behind data reconstruction is $Y=B^{-1}\times(W-A\times X)$, where Y is the lost original data, B^{-1} is the solution matrix, W is the surviving check data, A is the partial check data encoding matrix (for the surviving check drives and the surviving data drives), and X is the surviving original data.

In step 620, the outer loop processes the next 64-byte chunk of data for each of the drives. Like Sequencer 1, the first surviving data drive is again handled specially since the partial check data $A\times X$ has to be initialized for each surviving check drive.

In step 630, the first middle loop is called, in which the partial check data $A\times X$ is initialized for each surviving check drive based on the first surviving data drive's 64 bytes of data. In this case, the Parallel Lookup Multiplier is called for each surviving check drive with the corresponding factor (from A) for the first surviving data drive.

In step 640, the second middle loop is called, in which the lost check data is initialized for each failed check drive. Using the same 64 bytes of the first surviving data drive (preserved across the calls to Parallel Lookup Multiplier in step 630), the Parallel Lookup Multiplier is again called, this time to initialize each of the failed check drive's check data to the corresponding component from the first surviving data drive. This completes the computations involving the first surviving data drive's 64 bytes of data, which were fetched with one access from main memory and preserved in the same four registers across steps 630 and 640.

Continuing with FIG. 6, in step 650, the third middle loop is called, which processes the other surviving data drives' corresponding 64-byte chunks of data. As with the first surviving data drive, each of the other surviving data drives is processed separately, loading the respective 64 bytes of data into four registers (preserved across calls to the Parallel Lookup Multiplier).

In step 660, the first inner loop is called, in which the partial check data $A\times X$ is updated for each surviving check drive based on the next surviving data drive's 64 bytes of data. In this case, the Parallel Lookup Multiplier is called for each surviving check drive with the corresponding factor (from A) for the next surviving data drive.

In step 670, the second inner loop is called, in which the lost check data is updated for each failed check drive. Using the same 64 bytes of the next surviving data drive (preserved across the calls to Parallel Lookup Multiplier in step 660), the Parallel Lookup Multiplier is again called, this time to update each of the failed check drive's check data by the corresponding component from the next surviving data drive. This completes the computations involving the next surviving data

22

drive's 64 bytes of data, which were fetched with one access from main memory and preserved in the same four registers across steps 660 and 670.

Next, in step 680, the computation of the partial check data $A\times X$ is complete, so the surviving check data W is added to this result (recall that $W-A\times X$ is equivalent to $W+A\times X$ in binary Galois Field arithmetic). This is done by the fourth middle loop, which for each surviving check drive adds the corresponding 64-byte component of surviving check data W to the (surviving) partial check data $A\times X$ (using the Parallel Adder) to produce the (lost) partial check data $W-A\times X$.

Continuing with FIG. 7, in step 690, the fifth middle loop is called, which performs the two dimensional matrix multiplication $B^{-1}\times(W-A\times X)$ to produce the lost original data Y. The calculation is performed one row at a time, for a total of F rows, initializing the row to the first term of the corresponding linear combination of the solution matrix B^{-1} and the lost partial check data $W-A\times X$ (using the Parallel Lookup Multiplier).

In step 700, the third inner loop is called, which completes the remaining F-1 terms of the corresponding linear combination (using the Parallel Lookup Multiplier on each term) from the fifth middle loop in step 690 and updates the running calculation (using the Parallel Adder) of the next row of $B^{-1}\times(W-A\times X)$. This completes the next row (and reconstructs the corresponding failed data drive's lost data) of lost original data Y, which can then be stored at an appropriate location.

In step 710, the fourth inner loop is called, in which the lost check data is updated for each failed check drive by the newly reconstructed lost data for the next failed data drive. Using the same 64 bytes of the next reconstructed lost data (preserved across calls to the Parallel Lookup Multiplier), the Parallel Lookup Multiplier is called to update each of the failed check drives' check data by the corresponding component from the next failed data drive. This completes the computations involving the next failed data drive's 64 bytes of reconstructed data, which were performed as soon as the data was reconstructed and without being stored and retrieved from main memory.

Finally, in step 720, the sixth middle loop is called. The lost check data has been regenerated, so in this step, the newly regenerated check data is stored at an appropriate location (if desired).

Aspects of the present invention can be also realized in other environments, such as two-byte quantities, each such two-byte quantity capable of taking on $2^{16}=65,536$ possible values, by using similar constructs (scaled accordingly) to those presented here. Such extensions would be readily apparent to one of ordinary skill in the art, so their details will be omitted for brevity of description.

Exemplary techniques and methods for doing the Galois field manipulation and other mathematics behind RAID error correcting codes are described in Appendix A, which contains a paper "Information Dispersal Matrices for RAID Error Correcting Codes" prepared for the present application. Multi-Core Considerations

What follows is an exemplary embodiment for optimizing or improving the performance of multi-core architecture systems when implementing the described erasure coding system routines. In multi-core architecture systems, each processor die is divided into multiple CPU cores, each with their own local caches, together with a memory (bus) interface and possible on-die cache to interface with a shared memory with other processor dies.

FIG. 8 illustrates a multi-core architecture system 100 having two processor dies 110 (namely, Die 0 and Die 1).

US 8,683,296 B2

23

Referring to FIG. 8, each die **110** includes four central processing units (CPUs or cores) **120** each having a local level 1 (L1) cache. Each core **120** may have separate functional units, for example, an x86 execution unit (for traditional instructions) and a SSE execution unit (for software designed for the newer SSE instruction set). An example application of these function units is that the x86 execution unit can be used for the RAID control logic software while the SSE execution unit can be used for the GF operation software. Each die **110** also has a level 2 (L2) cache/memory bus interface **130** shared between the four cores **120**. Main memory **140**, in turn, is shared between the two dies **110**, and is connected to the input/output (I/O) controllers **150** that access external devices such as disk drives or other non-volatile storage devices via interfaces such as Peripheral Component Interconnect (PCI).

Redundant array of independent disks (RAID) controller processing can be described as a series of states or functions. These states may include: (1) Command Processing, to validate and schedule a host request (for example, to load or store data from disk storage); (2) Command Translation and Submission, to translate the host request into multiple disk requests and to pass the requests to the physical disks; (3) Error Correction, to generate check data and reconstruct lost data when some disks are not functioning correctly; and (4) Request Completion, to move data from internal buffers to requestor buffers. Note that the final state, Request Completion, may only be needed for a RAID controller that supports caching, and can be avoided in a cacheless design.

Parallelism is achieved in the embodiment of FIG. 8 by assigning different cores **120** to different tasks. For example, some of the cores **120** can be “command cores,” that is, assigned to the I/O operations, which includes reading and storing the data and check bytes to and from memory **140** and the disk drives via the I/O interface **150**. Others of the cores **120** can be “data cores,” and assigned to the GF operations, that is, generating the check data from the original data, reconstructing the lost data from the surviving data, etc., including the Parallel Lookup Multiplier and the sequencers described above. For example, in exemplary embodiments, a scheduler can be used to divide the original data **D** into corresponding portions of each block, which can then be processed independently by different cores **120** for applications such as check data generation and lost data reconstruction.

One of the benefits of this data core/command core subdivision of processing is ensuring that different code will be executed in different cores **120** (that is, command code in command cores, and data code in data cores). This improves the performance of the associated L1 cache in each core **120**, and avoids the “pollution” of these caches with code that is less frequently executed. In addition, empirical results show that the dies **110** perform best when only one core **120** on each die **110** does the GF operations (i.e., Sequencer **1** or Sequencer **2**, with corresponding calls to Parallel Lookup Multiplier) and the other cores **120** do the I/O operations. This helps localize the Parallel Lookup Multiplier code and associated data to a single core **120** and not compete with other cores **120**, while allowing the other cores **120** to keep the data moving between memory **140** and the disk drives via the I/O interface **150**.

Embodiments of the present invention yield scalable, high performance RAID systems capable of outperforming other systems, and at much lower cost, due to the use of high volume commodity components that are leveraged to achieve the result. This combination can be achieved by utilizing the mathematical techniques and code optimizations described elsewhere in this application with careful placement of the

24

resulting code on specific processing cores. Embodiments can also be implemented on fewer resources, such as single-core dies and/or single-die systems, with decreased parallelism and performance optimization.

The process of subdividing and assigning individual cores **120** and/or dies **110** to inherently parallelizable tasks will result in a performance benefit. For example, on a Linux system, software may be organized into “threads,” and threads may be assigned to specific CPUs and memory systems via the `kthread_bind` function when the thread is created. Creating separate threads to process the GF arithmetic allows parallel computations to take place, which multiplies the performance of the system.

Further, creating multiple threads for command processing allows for fully overlapped execution of the command processing states. One way to accomplish this is to number each command, then use the arithmetic MOD function (`%` in C language) to choose a separate thread for each command. Another technique is to subdivide the data processing portion of each command into multiple components, and assign each component to a separate thread.

FIG. 9 shows an exemplary disk drive configuration **200** according to an embodiment of the present invention.

Referring to FIG. 9, eight disks are shown, though this number can vary in other embodiments. The disks are divided into three types: data drives **210**, parity drive **220**, and check drives **230**. The eight disks break down as three data drives **210**, one parity drive **220**, and four check drives **230** in the embodiment of FIG. 9.

Each of the data drives **210** is used to hold a portion of data. The data is distributed uniformly across the data drives **210** in stripes, such as 192 KB stripes. For example, the data for an application can be broken up into stripes of 192 KB, and each of the stripes in turn broken up into three 64 KB blocks, each of the three blocks being written to a different one of the three data drives **210**.

The parity drive **220** is a special type of check drive in that the encoding of its data is a simple summation (recall that this is exclusive OR in binary GF arithmetic) of the corresponding bytes of each of the three data drives **210**. That is, check data generation (Sequencer **1**) or regeneration (Sequencer **2**) can be performed for the parity drive **220** using the Parallel Adder (and not the Parallel Lookup Multiplier). Accordingly, the check data for the parity drive **220** is relatively straightforward to build. Likewise, when one of the data drives **210** no longer functions correctly, the parity drive **220** can be used to reconstruct the lost data by adding (same as subtracting in binary GF arithmetic) the corresponding bytes from each of the two remaining data drives **210**. Thus, a single drive failure of one of the data drives **210** is very straightforward to handle when the parity drive **220** is available (no Parallel Lookup Multiplier). Accordingly, the parity drive **220** can replace much of the GF multiplication operations with GF addition for both check data generation and lost data reconstruction.

Each of the check drives **230** contains a linear combination of the corresponding bytes of each of the data drives **210**. The linear combination is different for each check drive **230**, but in general is represented by a summation of different multiples of each of the corresponding bytes of the data drives **210** (again, all arithmetic being GF arithmetic). For example, for the first check drive **230**, each of the bytes of the first data drive **210** could be multiplied by 4, each of the bytes of the second data drive **210** by 3, and each of the bytes of the third data drive **210** by 6, then the corresponding products for each of the corresponding bytes could be added to produce the first check drive data. Similar linear combinations could be used to produce the check drive data for the other check drives **230**.

The specifics of which multiples for which check drive are explained in Appendix A.

With the addition of the parity drive **220** and check drives **230**, eight drives are used in the RAID system **200** of FIG. 9. Accordingly, each 192 KB of original data is stored as 512 KB (i.e., eight blocks of 64 KB) of (original plus check) data. Such a system **200**, however, is capable of recovering all of the original data provided any three of these eight drives survive. That is, the system **200** can withstand a concurrent failure of up to any five drives and still preserve all of the original data.

Exemplary Routines to Implement an Embodiment

The error correcting code (ECC) portion of an exemplary embodiment of the present invention may be written in software as, for example, four functions, which could be named as ECCInitialize, ECCSolve, ECCGenerate, and ECCRegenerate. The main functions that perform work are ECCGenerate and ECCRegenerate. ECCGenerate generates check codes for data that are used to recover data when a drive suffers an outage (that is, ECCGenerate generates the check data J from the original data D using Sequencer 1). ECCRegenerate uses these check codes and the remaining data to recover data after such an outage (that is, ECCRegenerate uses the surviving check data W, the surviving original data X, and Sequencer 2 to reconstruct the lost original data Y while also regenerating any of the lost check data). Prior to calling either of these functions, ECCSolve is called to compute the constants used for a particular configuration of data drives, check drives, and failed drives (for example, ECCSolve builds the solution matrix B^{-1} together with the lists of surviving and failed data and check drives). Prior to calling ECCSolve, ECCInitialize is called to generate constant tables used by all of the other functions (for example, ECCInitialize builds the IDM E and the two lookup tables for the Parallel Lookup Multiplier).

ECCInitialize

The function ECCInitialize creates constant tables that are used by all subsequent functions. It is called once at program initialization time. By copying or precomputing these values up front, these constant tables can be used to replace more time-consuming operations with simple table look-ups (such as for the Parallel Lookup Multiplier). For example, four tables useful for speeding up the GF arithmetic include:

1. mvct—an array of constants used to perform GF multiplication with the PSHUF8 instruction that operates on SSE registers (that is, the Parallel Lookup Multiplier).

2. mast—contains the master encoding matrix S (or the Information Dispersal Matrix (IDM) E, as described in Appendix A), or at least the nontrivial portion, such as the check drive encoding matrix H

3. mul_tab—contains the results of all possible GF multiplication operations of any two operands (for example, $256 \times 256 = 65,536$ bytes for all of the possible products of two different one-byte quantities)

4. div_tab—contains the results of all possible GF division operations of any two operands (can be similar in size to mul_tab)

ECCSolve

The function ECC Solve creates constant tables that are used to compute a solution for a particular configuration of data drives, check drives, and failed drives. It is called prior to using the functions ECCGenerate or ECCRegenerate. It allows the user to identify a particular case of failure by describing the logical configuration of data drives, check drives, and failed drives. It returns the constants, tables, and lists used to either generate check codes or regenerate data.

For example, it can return the matrix B that needs to be inverted as well as the inverted matrix B^{-1} (i.e., the solution matrix).

ECCGenerate

The function ECCGenerate is used to generate check codes (that is, the check data matrix J) for a particular configuration of data drives and check drives, using Sequencer 1 and the Parallel Lookup Multiplier as described above. Prior to calling ECCGenerate, ECCSolve is called to compute the appropriate constants for the particular configuration of data drives and check drives, as well as the solution matrix B^{-1} .

ECCRegenerate

The function ECCRegenerate is used to regenerate data vectors and check code vectors for a particular configuration of data drives and check drives (that is, reconstructing the original data matrix D from the surviving data matrix X and the surviving check matrix W, as well as regenerating the lost check data from the restored original data), this time using Sequencer 2 and the Parallel Lookup Multiplier as described above. Prior to calling ECCRegenerate, ECCSolve is called to compute the appropriate constants for the particular configuration of data drives, check drives, and failed drives, as well as the solution matrix B^{-1} .

Exemplary Implementation Details

As discussed in Appendix A, there are two significant sources of computational overhead in erasure code processing (such as an erasure coding system used in RAID processing): the computation of the solution matrix B^{-1} for a given failure scenario, and the byte-level processing of encoding the check data J and reconstructing the lost data after a lost packet (e.g., data drive failure). By reducing the solution matrix B^{-1} to a matrix inversion of a $F \times F$ matrix, where F is the number of lost packets (e.g., failed drives), that portion of the computational overhead is for all intents and purposes negligible compared to the megabytes (MB), gigabytes (GB), and possibly terabytes (TB) of data that needs to be encoded into check data or reconstructed from the surviving original and check data. Accordingly, the remainder of this section will be devoted to the byte-level encoding and regenerating processing.

As already mentioned, certain practical simplifications can be assumed for most implementations. By using a Galois field of 256 entries, byte-level processing can be used for all of the GF arithmetic. Using the master encoding matrix S described in Appendix A, any combination of up to 127 data drives, 1 parity drive, and 128 check drives can be supported with such a Galois field. While, in general, any combination of data drives and check drives that adds up to 256 total drives is possible, not all combinations provide a parity drive when computed directly. Using the master encoding matrix S, on the other hand, allows all such combinations (including a parity drive) to be built (or simply indexed) from the same such matrix. That is, the appropriate sub-matrix (including the parity drive) can be used for configurations of less than the maximum number of drives.

In addition, using the master encoding matrix S permits further data drives and/or check drives can be added without requiring the recomputing of the IDM E (unlike other proposed solutions, which recompute E for every change of N or M). Rather, additional indexing of rows and/or columns of the master encoding matrix S will suffice. As discussed above, the use of the parity drive can eliminate or significantly reduce the somewhat complex GF multiplication operations associated with the other check drives and replaces them with simple GF addition (bitwise exclusive OR in binary Galois fields) operations. It should be noted that master encoding matrices with the above properties are possible for any power-

of-two number of drives $2^P = N_{max} + M_{max}$ where the maximum number of data drives N_{max} is one less than a power of two (e.g., $N_{max} = 127$ or 63) and the maximum number of check drives M_{max} (including the parity drive) is $2^P - N_{max}$.

As discussed earlier, in an exemplary embodiment of the present invention, a modern x86 architecture is used (being readily available and inexpensive). In particular, this architecture supports 16 XMM registers and the SSE instructions. Each XMM register is 128 bits and is available for special purpose processing with the SSE instructions. Each of these XMM registers holds 16 bytes (8-bit), so four such registers can be used to store 64 bytes of data. Thus, by using SSE instructions (some of which work on different operand sizes, for example, treating each of the XMM registers as containing 16 one-byte operands), 64 bytes of data can be operated at a time using four consecutive SSE instructions (e.g., fetching from memory, storing into memory, zeroing, adding, multiplying), the remaining registers being used for intermediate results and temporary storage. With such an architecture, several routines are useful for optimizing the byte-level performance, including the Parallel Lookup Multiplier, Sequencer 1, and Sequencer 2 discussed above.

While the above description contains many specific embodiments of the invention, these should not be construed as limitations on the scope of the invention, but rather as examples of specific embodiments thereof. Accordingly, the scope of the invention should be determined not by the embodiments illustrated, but by the appended claims and their equivalents.

Glossary of Some Variables

A	encoding matrix ($F \times K$), sub-matrix of T
B	encoding matrix ($F \times F$), sub-matrix of T
B^{-1}	solution matrix ($F \times F$)
C	encoded data matrix $((N + M) \times L) = \begin{bmatrix} D \\ J \end{bmatrix}$
C'	surviving encoded data matrix $(N \times L) = \begin{bmatrix} X \\ W \end{bmatrix}$
D	original data matrix ($N \times L$)
D'	permuted original data matrix $(N \times L) = \begin{bmatrix} X \\ Y \end{bmatrix}$
E	information dispersal matrix $(IDM)((N + M) \times N) = \begin{bmatrix} I_N \\ H \end{bmatrix}$
F	number of failed data drives
G	number of failed check drives
H	check drive encoding matrix ($M \times N$)
I	identity matrix ($I_K = K \times K$ identity matrix, $I_N = N \times N$ identity matrix)
J	encoded check data matrix ($M \times L$)
K	number of surviving data drives = $N - F$
L	data block size (elements or bytes)
M	number of check drives
M_{max}	maximum value of M
N	number of data drives
N_{max}	maximum value of N
O	zero matrix ($K \times F$), sub-matrix of T
S	master encoding matrix $((M_{max} + N_{max}) \times N_{max})$
T	transformed $IDM(N \times N) = \begin{bmatrix} I_K & O \\ A & B \end{bmatrix}$
W	surviving check data matrix ($F \times L$)

-continued

Glossary of Some Variables

X	surviving original data matrix ($K \times L$)
Y	lost original data matrix ($F \times L$)

What is claimed is:

1. A system for accelerated error-correcting code (ECC) processing comprising:
 - a processing core for executing computer instructions and accessing data from a main memory; and
 - a non-volatile storage medium for storing the computer instructions,
 wherein the processing core, the non-volatile storage medium, and the computer instructions are configured to implement an erasure coding system comprising:
 - a data matrix for holding original data in the main memory;
 - a check matrix for holding check data in the main memory;
 - an encoding matrix for holding first factors in the main memory, the first factors being for encoding the original data into the check data; and
 - a thread for executing on the processing core and comprising:
 - a parallel multiplier for concurrently multiplying multiple data entries of a matrix by a single factor; and
 - a first sequencer for ordering operations through the data matrix and the encoding matrix using the parallel multiplier to generate the check data.
2. The system of claim 1, wherein the first sequencer is configured to access each entry of the data matrix from the main memory at most once while generating the check data.
3. The system of claim 1, wherein:
 - the processing core comprises a plurality of processing cores;
 - the thread comprises a plurality of threads; and
 - the erasure coding system further comprises a scheduler for generating the check data by:
 - dividing the data matrix into a plurality of data matrices;
 - dividing the check matrix into a plurality of check matrices;
 - assigning corresponding ones of the data matrices and the check matrices to the threads; and
 - assigning the threads to the processing cores to concurrently generate portions of the check data corresponding to the check matrices from respective ones of the data matrices.
4. The system of claim 1, wherein:
 - the data matrix comprises a first number of rows;
 - the check matrix comprises a second number of rows; and
 - the encoding matrix comprises the second number of rows and the first number of columns.
5. The system of claim 4, wherein the data matrix is configured to add rows to the first number of rows or the check matrix is configured to add rows to the second number of rows while the first factors remain unchanged.
6. The system of claim 4, wherein each of entries of one of the rows of the encoding matrix comprise a multiplicative identity factor.
7. The system of claim 4, wherein:
 - the data matrix is configured to be divided by rows into:
 - a surviving data matrix for holding surviving original data of the original data; and

US 8,683,296 B2

29

a lost data matrix corresponding to lost original data of the original data and comprising a third number of rows; and
the erasure coding system further comprises a solution matrix for holding second factors in the main memory, the second factors being for decoding the check data into the lost original data using the surviving original data and the first factors.

8. The system of claim 7, wherein the solution matrix comprises the third number of rows and the third number of columns.

9. The system of claim 8, wherein the solution matrix further comprises an inverted said third number by said third number sub-matrix of the encoding matrix.

10. The system of claim 7, wherein the erasure coding system further comprises:

a first list of rows of the data matrix corresponding to the surviving data matrix; and
a second list of rows of the data matrix corresponding to the lost data matrix.

11. The system of claim 1, wherein:

the data matrix is configured to be divided into:

a surviving data matrix for holding surviving original data of the original data; and
a lost data matrix corresponding to lost original data of the original data;

the erasure coding system further comprises a solution matrix for holding second factors in the main memory, the second factors being for decoding the check data into the lost original data using the surviving original data and the first factors; and

the thread further comprises a second sequencer for ordering operations through the surviving data matrix, the encoding matrix, the check matrix, and the solution matrix using the parallel multiplier to reconstruct the lost original data.

12. The system of claim 11, wherein the second sequencer is further configured to access each entry of the surviving data matrix from the main memory at most once while reconstructing the lost original data.

13. The system of claim 11, wherein:

the processing core comprises a plurality of processing cores;

the thread comprises a plurality of threads; and

the erasure coding system further comprises a scheduler for generating the check data and reconstructing the lost original data by:

dividing the data matrix into a plurality of data matrices;
dividing the surviving data matrix into a plurality of surviving data matrices;
dividing the lost data matrix into a plurality of lost data matrices;
dividing the check matrix into a plurality of check matrices;

assigning corresponding ones of the data matrices, the surviving data matrices, the lost data matrices, and the check matrices to the threads; and

assigning the threads to the processing cores to concurrently generate portions of the check data corresponding to the check matrices from respective ones of the data matrices and to concurrently reconstruct portions of the lost original data corresponding to the lost data matrices from respective ones of the surviving data matrices and the check matrices.

30

14. The system of claim 11, wherein:

the check matrix is configured to be divided into:

a surviving check matrix for holding surviving check data of the check data; and

a lost check matrix corresponding to lost check data of the check data; and

the second sequencer is configured to order operations through the surviving data matrix, the reconstructed lost original data, and the encoding matrix using the parallel multiplier to regenerate the lost check data.

15. The system of claim 14, wherein the second sequencer is further configured to reconstruct the lost original data concurrently with regenerating the lost check data.

16. The system of claim 15, wherein the second sequencer is further configured to access each entry of the surviving data matrix from the main memory at most once while reconstructing the lost original data and regenerating the lost check data.

17. The system of claim 15, wherein the second sequencer is further configured to regenerate the lost check data without accessing the reconstructed lost original data from the main memory.

18. The system of claim 14, wherein:

the processing core comprises a plurality of processing cores;

the thread comprises a plurality of threads; and

the erasure coding system further comprises a scheduler for generating the check data, reconstructing the lost original data, and regenerating the lost check data by:

dividing the data matrix into a plurality of data matrices;
dividing the surviving data matrix into a plurality of surviving data matrices;

dividing the lost data matrix into a plurality of lost data matrices;

dividing the check matrix into a plurality of check matrices;

dividing the surviving check matrix into a plurality of surviving check matrices;

dividing the lost check matrix into a plurality of lost check matrices;

assigning corresponding ones of the data matrices, the surviving data matrices, the lost data matrices, the check matrices, the surviving check matrices, and the lost check matrices to the threads; and

assigning the threads to the processing cores to concurrently generate portions of the check data corresponding to the check matrices from respective ones of the data matrices, to concurrently reconstruct portions of the lost original data corresponding to the lost data matrices from respective ones of the surviving data matrices and the surviving check matrices, and to concurrently regenerate portions of the lost check data corresponding to the lost check matrices from respective ones of the surviving data matrices and respective portions of the reconstructed lost original data.

19. The system of claim 1, wherein the processing core comprises 16 data registers, each of the data registers comprises 16 bytes; and the parallel multiplier is configured to process the data in units of at least 64 bytes spread over at least four of the data registers at a time.

20. The system of claim 19, wherein consecutive instructions to process each of the units of the data access separate ones of the data registers to permit concurrent execution of the consecutive instructions by the processing core.

21. The system of claim 19, wherein the parallel multiplier comprises two lookup tables for doing concurrent multipli-

US 8,683,296 B2

31

cation of 4-bit quantities across 16 byte-sized entries using the PSHUFB (Packed Shuffle Bytes) instruction.

22. The system of claim 19, wherein the parallel multiplier is further configured to:

receive an input operand in four of the data registers; and
return with the input operand intact in the four of the data registers.

23. A method of accelerated error-correcting code (ECC) processing on a computing system comprising a non-volatile storage medium, a processing core for accessing instructions and data from a main memory, and a computer program comprising a plurality of computer instructions for implementing an erasure coding system, the method comprising:

storing the computer program on the non-volatile storage medium;
executing the computer instructions on the processing core;

arranging original data as a data matrix in the main memory;

arranging first factors as an encoding matrix in the main memory, the first factors being for encoding the original data into check data, the check data being arranged as a check matrix in the main memory; and

generating the check data using a parallel multiplier for concurrently multiplying multiple data entries of a matrix by a single factor, the generating of the check data comprising ordering operations through the data matrix and the encoding matrix using the parallel multiplier.

24. The method of claim 23, wherein the generating of the check data comprises accessing each entry of the data matrix from the main memory at most once.

25. The method of claim 23, wherein:

the processing core comprises a plurality of processing cores;

the executing of the computer instructions comprises executing the computer instructions on the processing cores;

the method further comprises scheduling the generating of the check data by:

dividing the data matrix into a plurality of data matrices;
dividing the check matrix into a plurality of check matrices; and

assigning corresponding ones of the data matrices and the check matrices to the processing cores to concurrently generate portions of the check data corresponding to the check matrices from respective ones of the data matrices.

26. The method of claim 23, further comprising:

dividing the data matrix into:

a surviving data matrix for holding surviving original data of the original data; and

a lost data matrix corresponding to lost original data of the original data;

arranging second factors as a solution matrix in the main memory, the second factors being for decoding the check data into the lost original data using the surviving original data and the first factors; and

reconstructing the lost original data by ordering operations through the surviving data matrix, the encoding matrix, the check matrix, and the solution matrix using the parallel multiplier.

27. The method of claim 26, wherein the reconstructing of the lost original data comprises accessing each entry of the surviving data matrix from the main memory at most once.

28. The method of claim 26, wherein:

the processing core comprises a plurality of processing cores;

32

the executing of the computer instructions comprises executing the computer instructions on the processing cores;

the method further comprises scheduling the generating of the check data and the reconstructing of the lost original data by:

dividing the data matrix into a plurality of data matrices;
dividing the surviving data matrix into a plurality of surviving data matrices;

dividing the lost data matrix into a plurality of lost data matrices;

dividing the check matrix into a plurality of check matrices; and

assigning corresponding ones of the data matrices, the surviving data matrices, the lost data matrices, and the check matrices to the processing cores to concurrently generate portions of the check data corresponding to the check matrices from respective ones of the data matrices and to concurrently reconstruct portions of the lost original data corresponding to the lost data matrices from respective ones of the surviving data matrices and the check matrices.

29. The method of claim 26, further comprising

dividing the check matrix into:

a surviving check matrix for holding surviving check data of the check data; and

a lost check matrix corresponding to lost check data of the check data; and

regenerating the lost check data by ordering operations through the surviving data matrix, the reconstructed lost original data, and the encoding matrix using the parallel multiplier.

30. The method of claim 29, wherein the reconstructing of the lost original data takes place concurrently with the regenerating of the lost check data.

31. The method of claim 30, wherein the reconstructing of the lost original data and the regenerating of the lost check data comprise accessing each entry of the surviving data matrix from the main memory at most once.

32. The method of claim 30, wherein the regenerating of the lost check data takes place without accessing the reconstructed lost original data from the main memory.

33. The method of claim 29, wherein:

the processing core comprises a plurality of processing cores;

the executing of the computer instructions comprises executing the computer instructions on the processing cores;

the method further comprises scheduling the generating of the check data, the reconstructing of the lost original data, and the regenerating of the lost check data by:

dividing the data matrix into a plurality of data matrices;
dividing the surviving data matrix into a plurality of surviving data matrices;

dividing the lost data matrix into a plurality of lost data matrices;

dividing the check matrix into a plurality of check matrices;

dividing the surviving check matrix into a plurality of surviving check matrices;

dividing the lost check matrix into a plurality of lost check matrices; and

assigning corresponding ones of the data matrices, the surviving data matrices, the lost data matrices, the check matrices, the surviving check matrices, and the lost check matrices to the processing cores to concurrently generate portions of the check data correspond-

US 8,683,296 B2

33

ing to the check matrices from respective ones of the data matrices, to concurrently reconstruct portions of the lost original data corresponding to the lost data matrices from respective ones of the surviving data matrices and the surviving check matrices, and to concurrently regenerate portions of the lost check data corresponding to the lost check matrices from respective ones of the surviving data matrices and respective portions of the reconstructed lost original data.

34. A non-transitory computer-readable storage medium containing a computer program comprising a plurality of computer instructions for performing accelerated error-correcting code (ECC) processing on a computing system comprising a processing core for accessing instructions and data from a main memory, the computer instructions being configured to implement an erasure coding system when executed on the computing system by performing the steps of:

arranging original data as a data matrix in the main memory;

arranging first factors as an encoding matrix in the main memory, the first factors being for encoding the original data into check data, the check data being arranged as a check matrix in the main memory; and

generating the check data using a parallel multiplier for concurrently multiplying multiple data entries of a matrix by a single factor, the generating of the check data comprising ordering operations through the data matrix and the encoding matrix using the parallel multiplier.

35. The storage medium of claim **34**, wherein the generating of the check data comprises accessing each entry of the data matrix from the main memory at most once.

36. The storage medium of claim **34**, wherein:

the processing core comprises a plurality of processing cores; and

the computer instructions are further configured to perform the step of scheduling the generating of the check data by:

dividing the data matrix into a plurality of data matrices; dividing the check matrix into a plurality of check matrices; and

assigning corresponding ones of the data matrices and the check matrices to the processing cores to concurrently generate portions of the check data corresponding to the check matrices from respective ones of the data matrices.

37. The storage medium of claim **34**, wherein the computer instructions are further configured to perform the steps of:

dividing the data matrix into:

a surviving data matrix for holding surviving original data of the original data; and

a lost data matrix corresponding to lost original data of the original data;

arranging second factors as a solution matrix in the main memory, the second factors being for decoding the

34

check data into the lost original data using the surviving original data and the first factors; and reconstructing the lost original data by ordering operations through the surviving data matrix, the encoding matrix, the check matrix, and the solution matrix using the parallel multiplier.

38. The storage medium of claim **37**, wherein the computer instructions are further configured to perform the steps of: dividing the check matrix into:

a surviving check matrix for holding surviving check data of the check data; and

a lost check matrix corresponding to lost check data of the check data; and

regenerating the lost check data by ordering operations through the surviving data matrix, the reconstructed lost original data, and the encoding matrix using the parallel multiplier.

39. The storage medium of claim **38**, wherein the reconstructing of the lost original data and the regenerating of the lost check data comprise accessing each entry of the surviving data matrix from the main memory at most once.

40. The storage medium of claim **38**, wherein:

the processing core comprises a plurality of processing cores;

the computer instructions are further configured to perform the step of scheduling the generating of the check data, the reconstructing of the lost original data, and the regenerating of the lost check data by:

dividing the data matrix into a plurality of data matrices; dividing the surviving data matrix into a plurality of surviving data matrices;

dividing the lost data matrix into a plurality of lost data matrices;

dividing the check matrix into a plurality of check matrices;

dividing the surviving check matrix into a plurality of surviving check matrices;

dividing the lost check matrix into a plurality of lost check matrices; and

assigning corresponding ones of the data matrices, the surviving data matrices, the lost data matrices, the check matrices, the surviving check matrices, and the lost check matrices to the processing cores to concurrently generate portions of the check data corresponding to the check matrices from respective ones of the data matrices, to concurrently reconstruct portions of the lost original data corresponding to the lost data matrices from respective ones of the surviving data matrices and the surviving check matrices, and to concurrently regenerate portions of the lost check data corresponding to the lost check matrices from respective ones of the surviving data matrices and respective portions of the reconstructed lost original data.

* * * * *

EXHIBIT B



US009160374B2

(12) **United States Patent**
Anderson

(10) **Patent No.:** **US 9,160,374 B2**
(45) **Date of Patent:** ***Oct. 13, 2015**

(54) **ACCELERATED ERASURE CODING SYSTEM AND METHOD**

(71) Applicant: **STREAMSCALE, INC.**, Los Angeles, CA (US)

(72) Inventor: **Michael H. Anderson**, Los Angeles, CA (US)

(73) Assignee: **STREAMSCALE, INC.**, Los Angeles, CA (US)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 0 days.

This patent is subject to a terminal disclaimer.

(21) Appl. No.: **14/223,740**

(22) Filed: **Mar. 24, 2014**

(65) **Prior Publication Data**

US 2015/0012796 A1 Jan. 8, 2015

Related U.S. Application Data

(63) Continuation of application No. 13/341,833, filed on Dec. 30, 2011, now Pat. No. 8,683,296.

(51) **Int. Cl.**
H03M 13/00 (2006.01)
H03M 13/37 (2006.01)
(Continued)

(52) **U.S. Cl.**
CPC **H03M 13/616** (2013.01); **G06F 11/1076** (2013.01); **G06F 11/1092** (2013.01);
(Continued)

(58) **Field of Classification Search**
CPC H03M 13/373; H03M 13/3761; H03M 13/3776; H03M 13/616; H03M 13/1191; H03M 13/134; H03M 13/1515; H04L 1/0043;

H04L 1/0057; G06F 11/1076; G06F 11/1092; G06F 11/1096; G06F 12/0238; G06F 12/06; G06F 2211/1057; G06F 2211/109
USPC 714/6.24, 6.1, 6.11, 6.2, 6.21, 6.32, 714/763, 752, 758, 768, 770, 773, 784, 786
See application file for complete search history.

(56) **References Cited**

U.S. PATENT DOCUMENTS

6,654,924 B1 * 11/2003 Hassner et al. 714/758
6,823,425 B2 * 11/2004 Ghosh et al. 711/114

(Continued)

OTHER PUBLICATIONS

Hafner et al., Matrix Methods for Lost Data Reconstruction in Erasure Codes, Nov. 16, 2005, USENIX FAST '05 Paper, pp. 1-26.*
Anvin; The mathematics of RAID-6; First Version Jan. 20, 2004; Last Updated Dec. 20, 2011; pp. 1-9.

Maddock, et al.; White Paper, Surviving Two Disk Failures Introducing Various "RAID 6" Implementations; Xyratex; pp. 1-13.

(Continued)

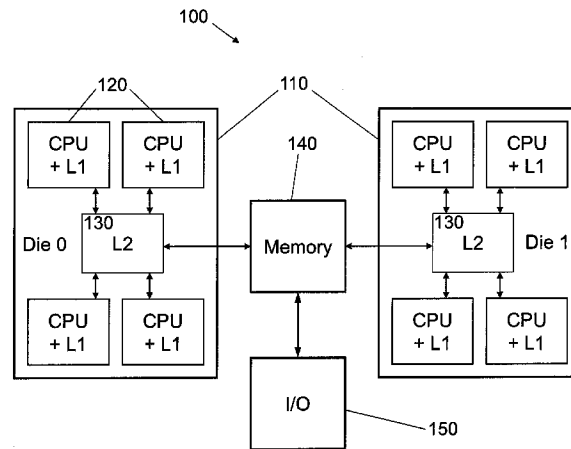
Primary Examiner — John J Tabone, Jr.

(74) *Attorney, Agent, or Firm* — Christie, Parker & Hale, LLP

(57) **ABSTRACT**

An accelerated erasure coding system includes a processing core for executing computer instructions and accessing data from a main memory, and a non-volatile storage medium for storing the computer instructions. The processing core, storage medium, and computer instructions are configured to implement an erasure coding system, which includes: a data matrix for holding original data in the main memory; a check matrix for holding check data in the main memory; an encoding matrix for holding first factors in the main memory, the first factors being for encoding the original data into the check data; and a thread for executing on the processing core. The thread includes: a parallel multiplier for concurrently multiplying multiple entries of the data matrix by a single entry of the encoding matrix; and a first sequencer for ordering operations through the data matrix and the encoding matrix using the parallel multiplier to generate the check data.

18 Claims, 9 Drawing Sheets



US 9,160,374 B2

Page 2

(51)	Int. Cl. <i>H03M 13/13</i> (2006.01) <i>H04L 1/00</i> (2006.01) <i>G06F 11/10</i> (2006.01) <i>G06F 12/02</i> (2006.01) <i>G06F 12/06</i> (2006.01) <i>H03M 13/15</i> (2006.01) <i>H03M 13/11</i> (2006.01)	(56)	References Cited U.S. PATENT DOCUMENTS 7,350,126 B2 * 3/2008 Winograd et al. 714/752 7,930,337 B2 4/2011 Hasenplaugh et al. 8,145,941 B2 * 3/2012 Jacobson 714/6.24 8,352,847 B2 * 1/2013 Gunnam 714/801 2011/0029756 A1 * 2/2011 Biscondi et al. 712/22 2012/0272036 A1 * 10/2012 Muralimanohar et al. ... 711/202 2013/0108048 A1 * 5/2013 Grube et al. 380/270 2013/0110962 A1 * 5/2013 Grube et al. 709/213 2013/0111552 A1 * 5/2013 Grube et al. 726/3 2013/0124932 A1 * 5/2013 Schuh et al. 714/718 2013/0173956 A1 * 7/2013 Anderson 714/6.24 OTHER PUBLICATIONS Plank; All About Erasure Codes:—Reed-Solomon Coding—LDPC Coding; Logistical Computing and Internetworking Laboratory, Department of Computer Science, University of Tennessee; ICL—Aug. 20, 2004; 52 sheets. * cited by examiner
(52)	U.S. Cl. CPC <i>G06F11/1096</i> (2013.01); <i>G06F 12/0238</i> (2013.01); <i>G06F 12/06</i> (2013.01); <i>H03M 13/1191</i> (2013.01); <i>H03M 13/134</i> (2013.01); <i>H03M 13/1515</i> (2013.01); <i>H03M 13/373</i> (2013.01); <i>H03M 13/3761</i> (2013.01); <i>H03M 13/3776</i> (2013.01); <i>H04L 1/0043</i> (2013.01); <i>H04L 1/0057</i> (2013.01); <i>G06F 2211/1057</i> (2013.01)		

FIG. 1

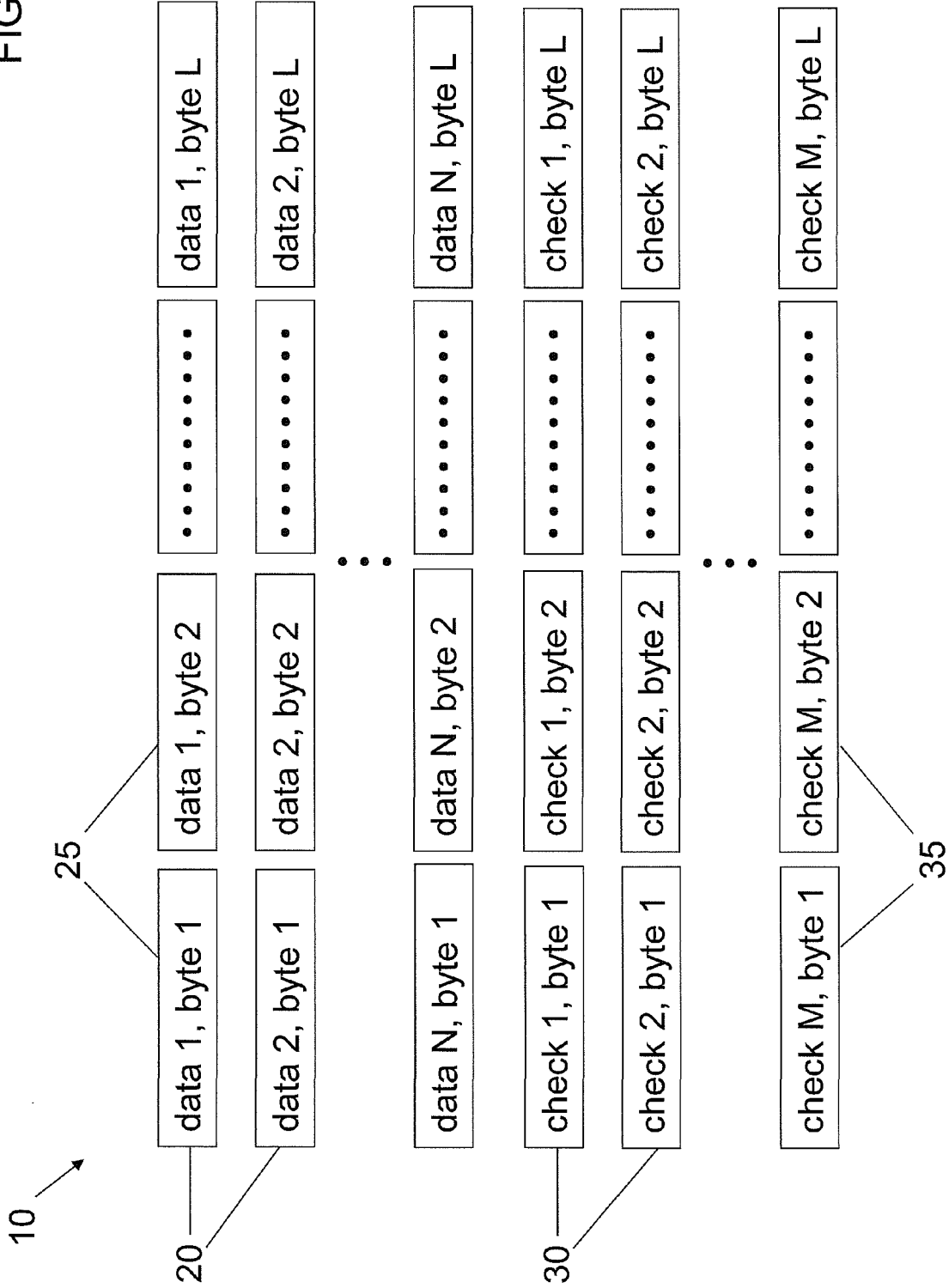
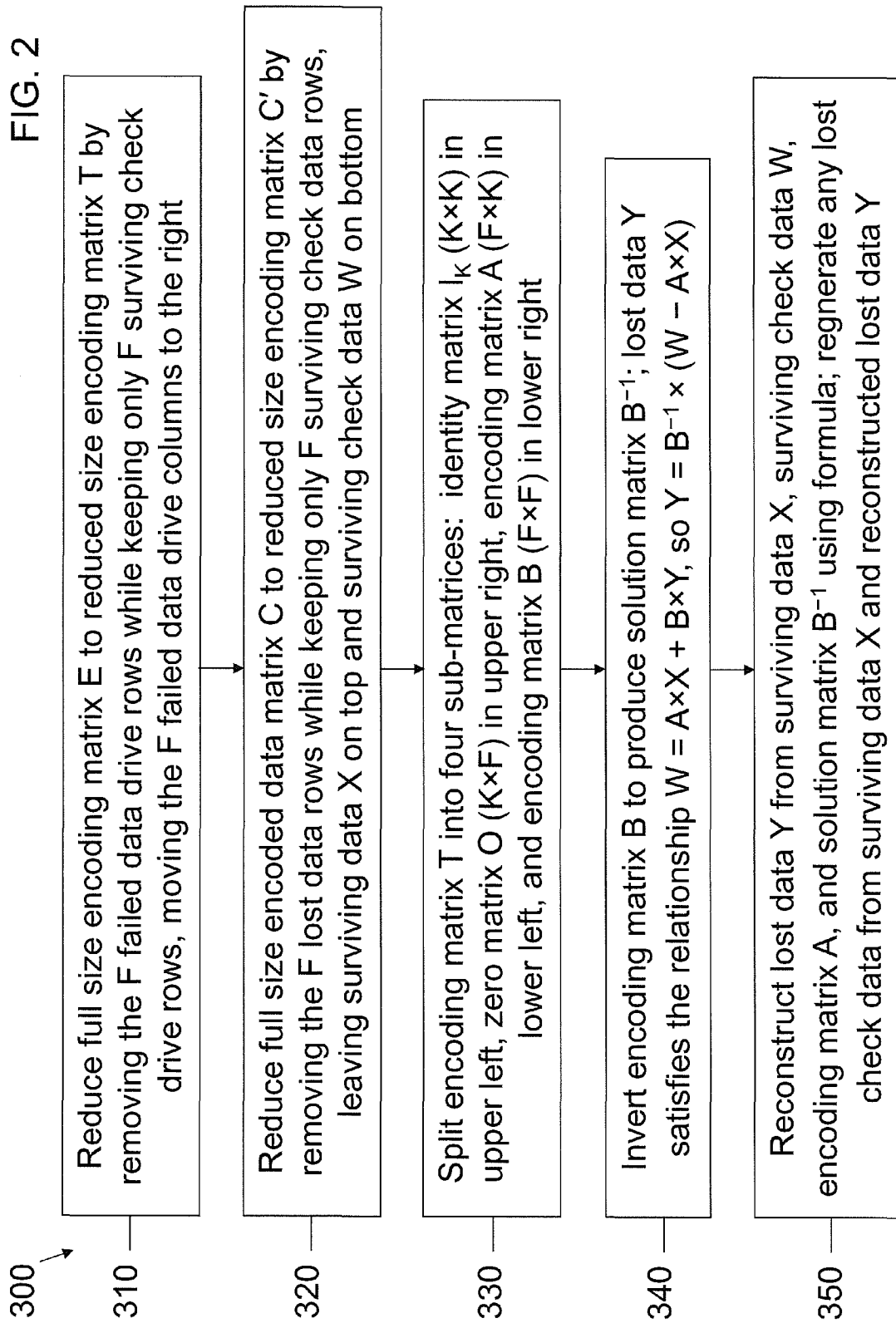


FIG. 2



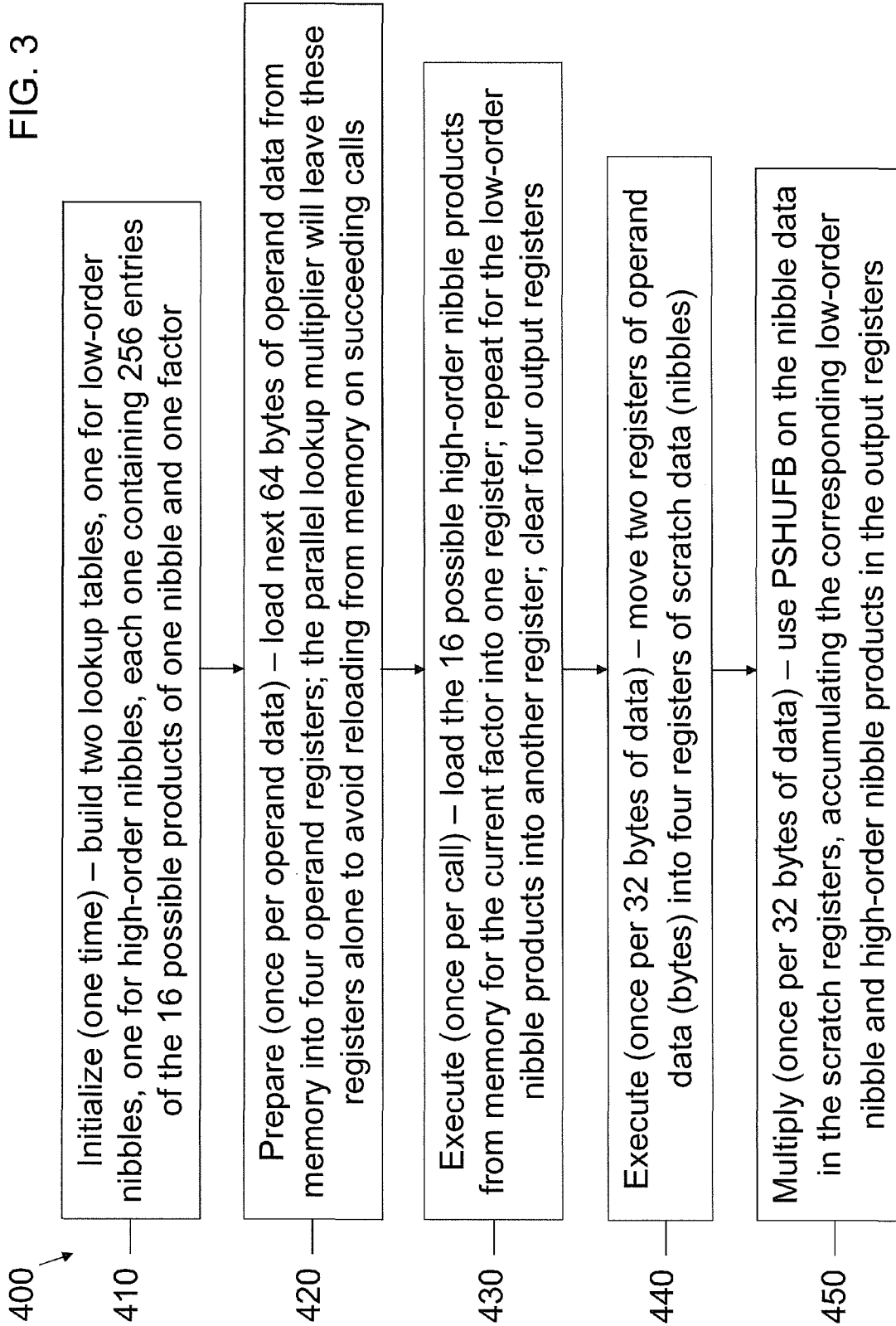


FIG. 4

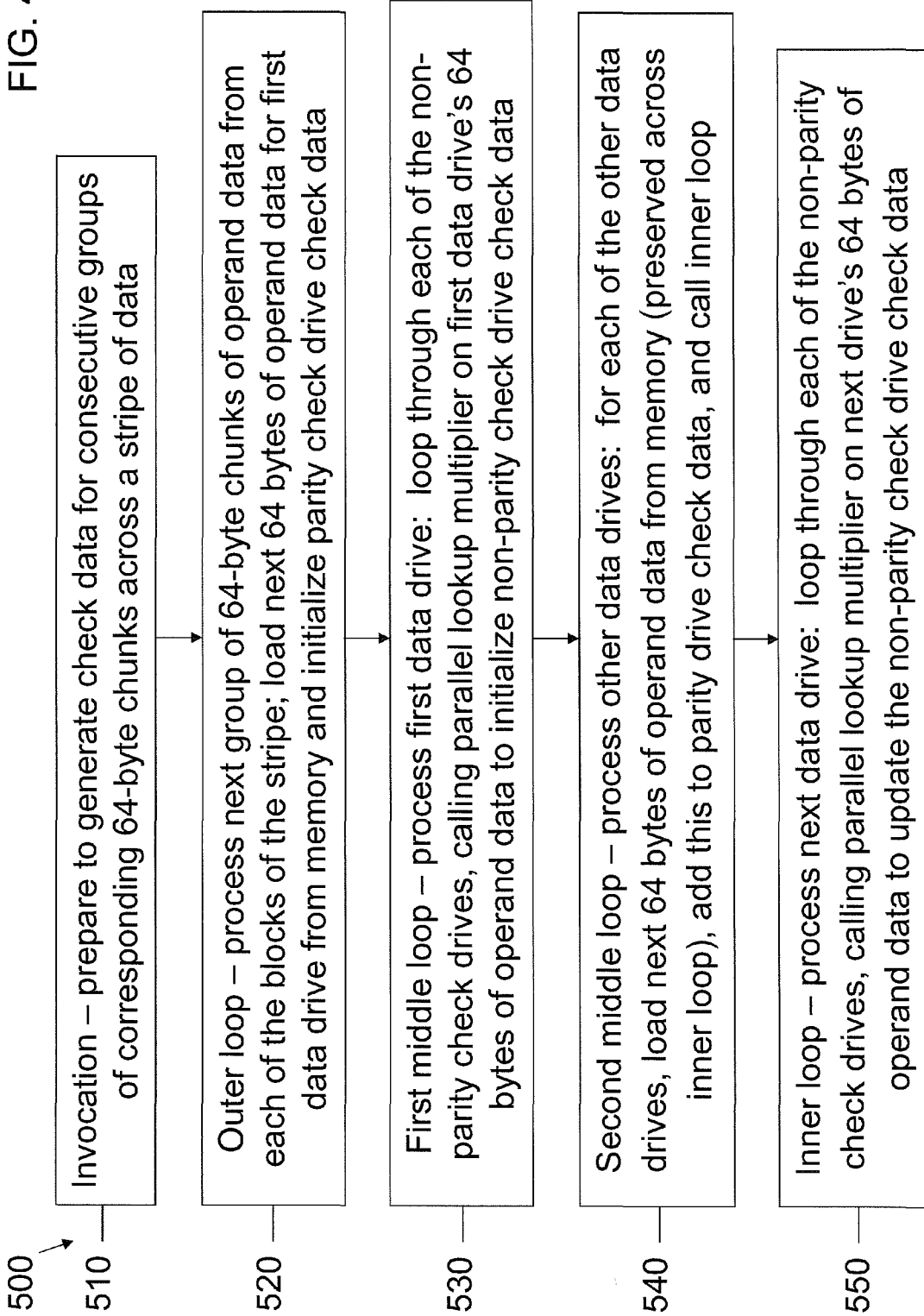


FIG. 5

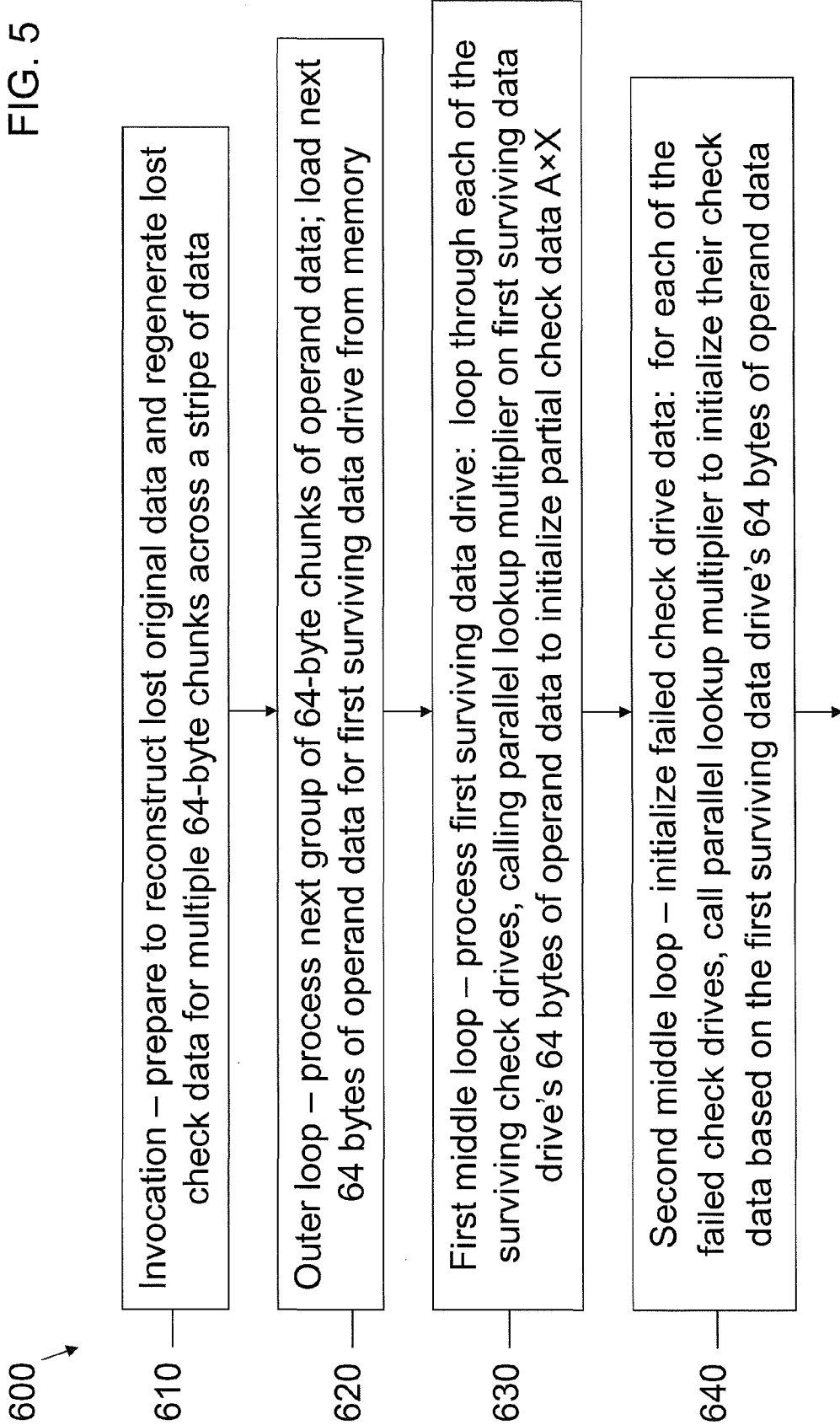


FIG. 6

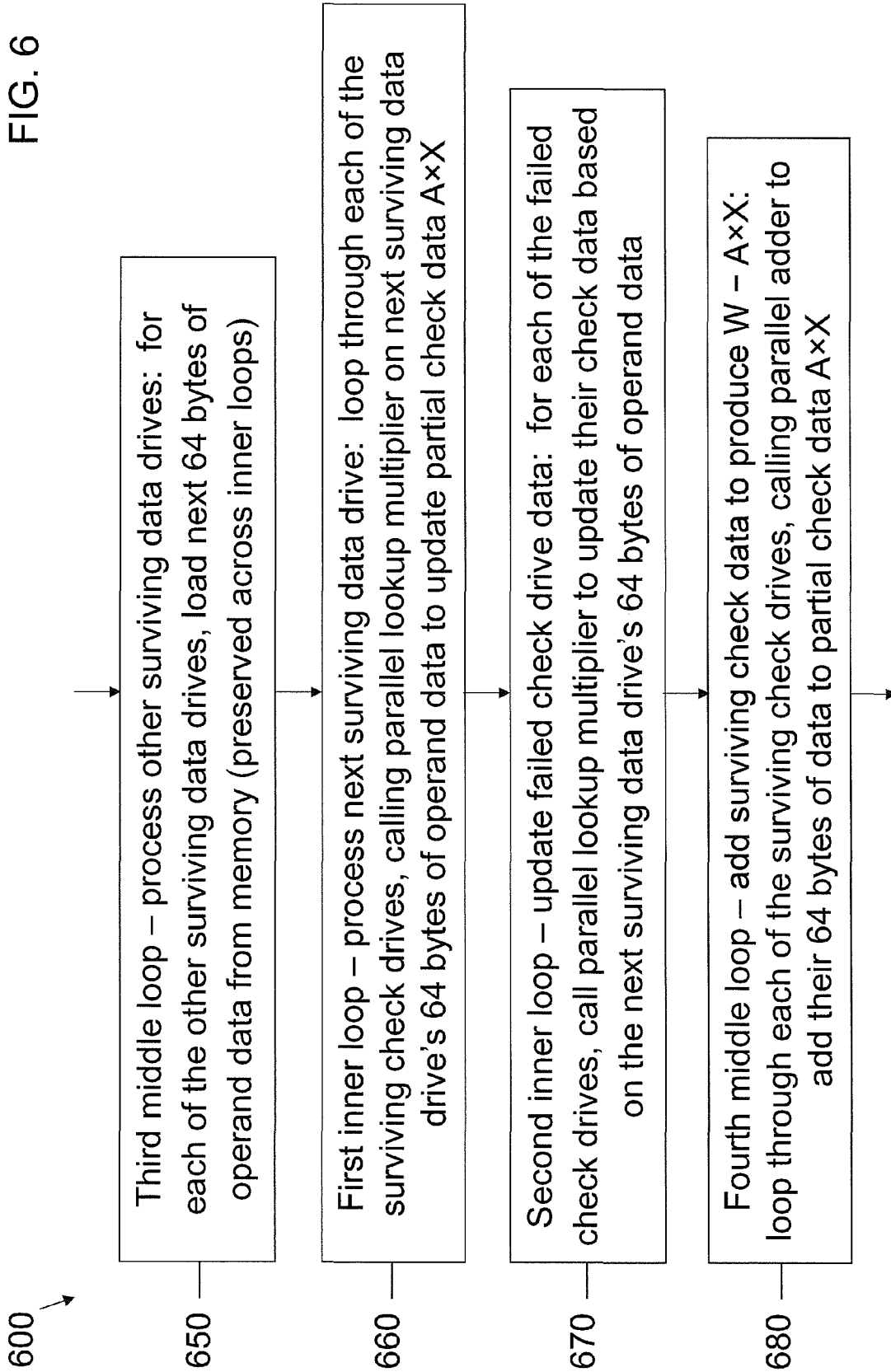


FIG. 7

600 →

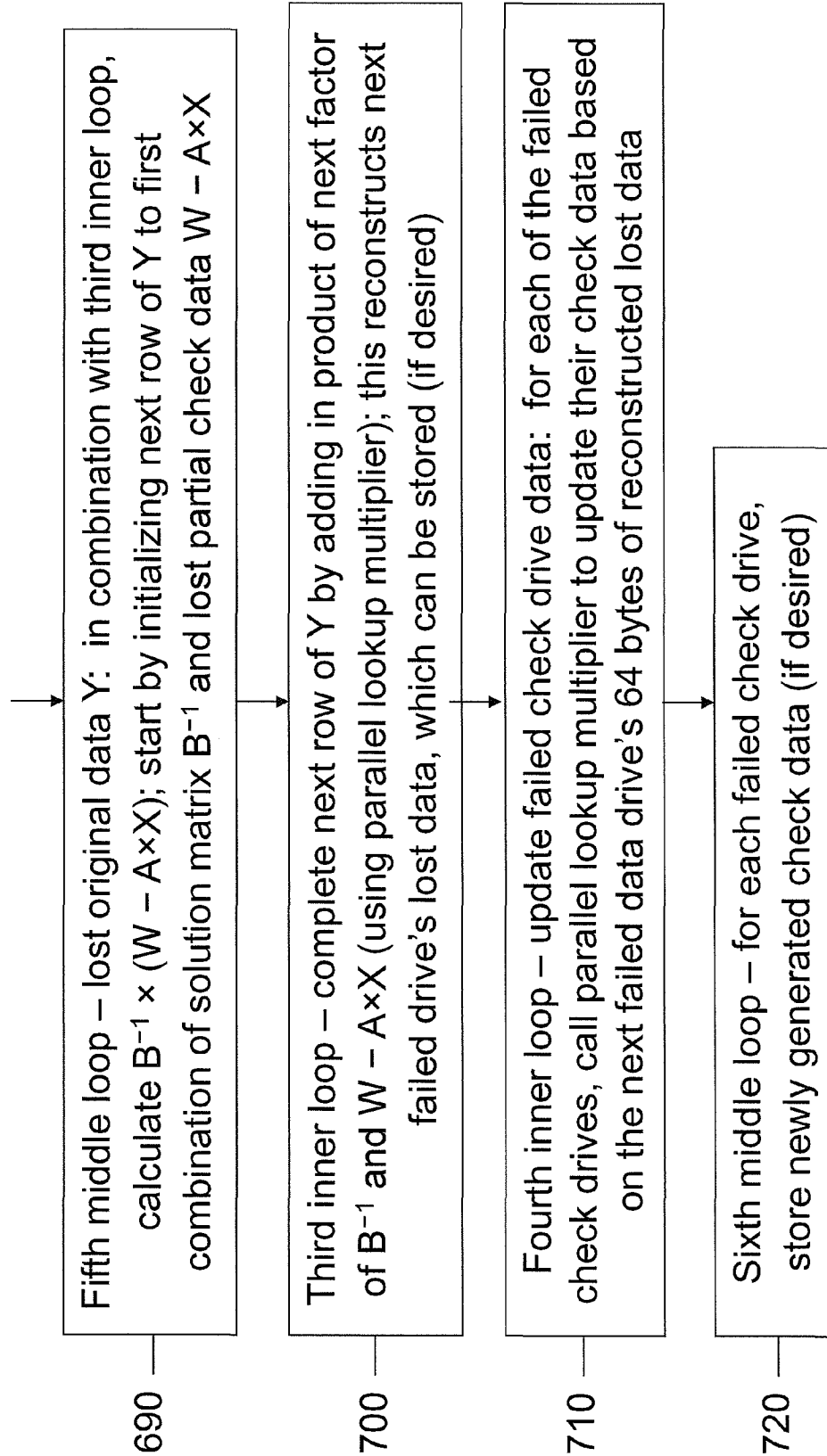


FIG. 8

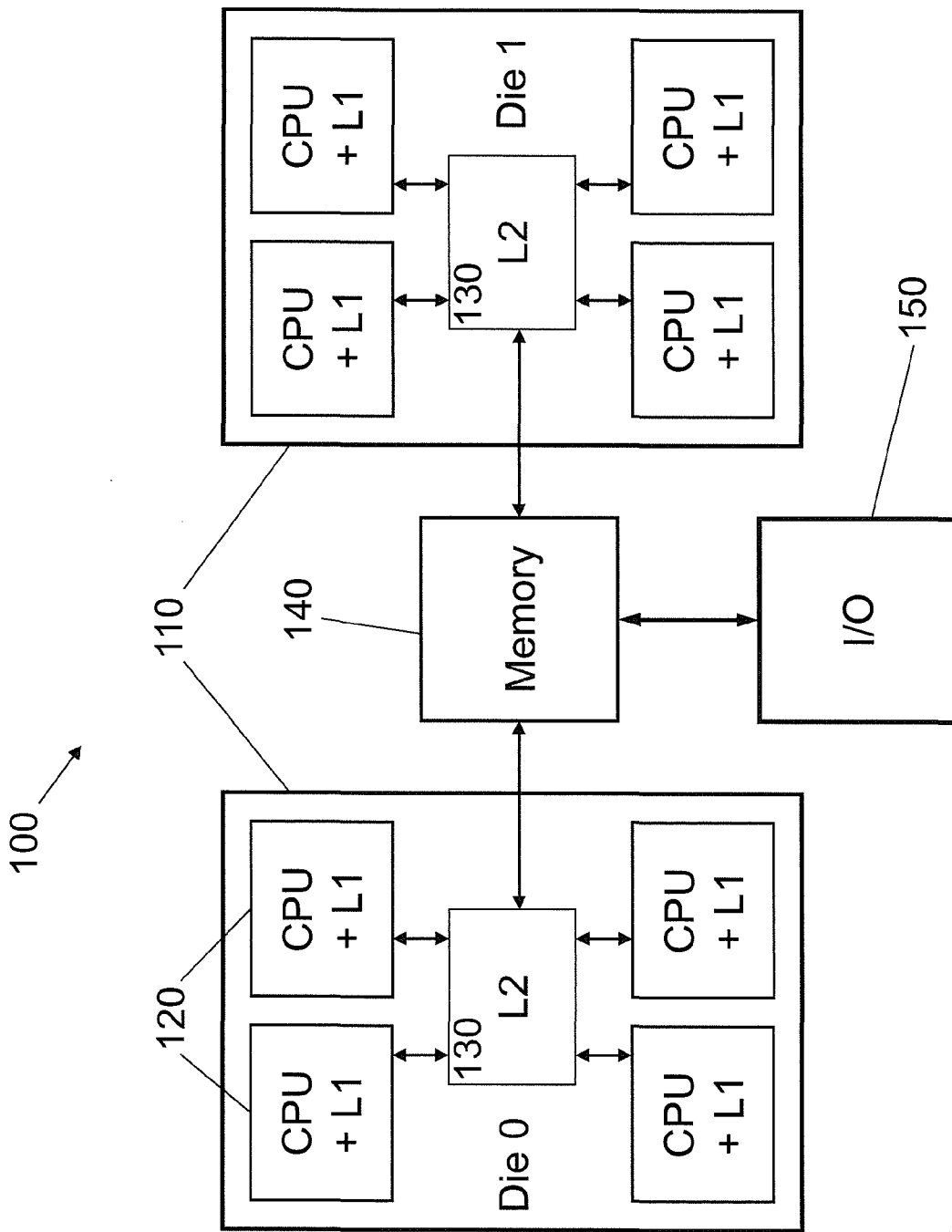
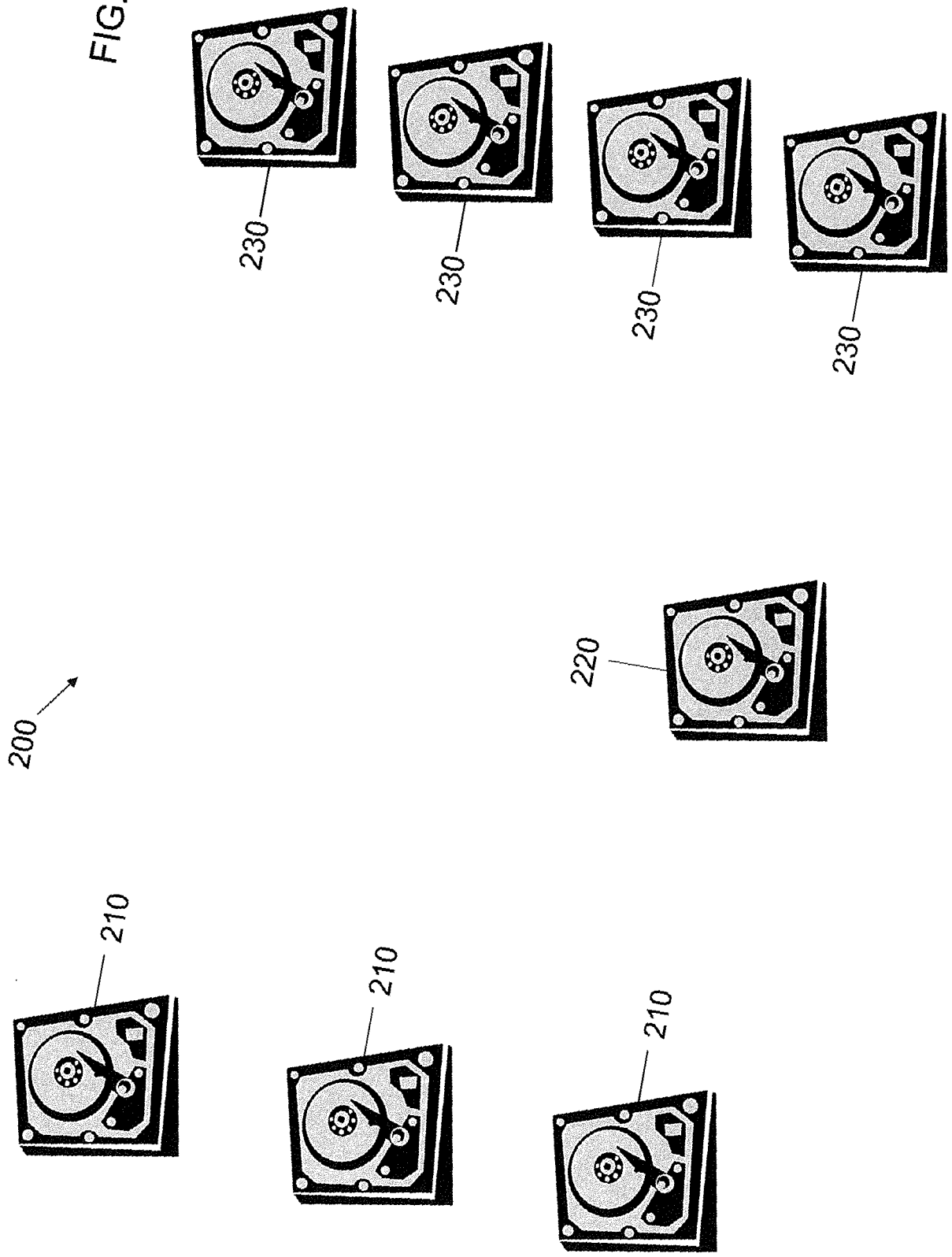


FIG. 9



US 9,160,374 B2

1

ACCELERATED ERASURE CODING SYSTEM AND METHOD

CROSS-REFERENCE TO RELATED APPLICATION

This application is a continuation of U.S. patent application Ser. No. 13/341,833, filed on Dec. 30, 2011, now U.S. Pat. No. 8,683,296, issued on Mar. 25, 2014, the entire contents of which is expressly incorporated herein by reference.

BACKGROUND

1. Field

Aspects of embodiments of the present invention are directed toward an accelerated erasure coding system and method.

2. Description of Related Art

An erasure code is a type of error-correcting code (ECC) useful for forward error-correction in applications like a redundant array of independent disks (RAID) or high-speed communication systems. In a typical erasure code, data (or original data) is organized in stripes, each of which is broken up into N equal-sized blocks, or data blocks, for some positive integer N . The data for each stripe is thus reconstructable by putting the N data blocks together. However, to handle situations where one or more of the original N data blocks gets lost, erasure codes also encode an additional M equal-sized blocks (called check blocks or check data) from the original N data blocks, for some positive integer M .

The N data blocks and the M check blocks are all the same size. Accordingly, there are a total of $N+M$ equal-sized blocks after encoding. The $N+M$ blocks may, for example, be transmitted to a receiver as $N+M$ separate packets, or written to $N+M$ corresponding disk drives. For ease of description, all $N+M$ blocks after encoding will be referred to as encoded blocks, though some (for example, N of them) may contain unencoded portions of the original data. That is, the encoded data refers to the original data together with the check data.

The M check blocks build redundancy into the system, in a very efficient manner, in that the original data (as well as any lost check data) can be reconstructed if any N of the $N+M$ encoded blocks are received by the receiver, or if any N of the $N+M$ disk drives are functioning correctly. Note that such an erasure code is also referred to as “optimal.” For ease of description, only optimal erasure codes will be discussed in this application. In such a code, up to M of the encoded blocks can be lost, (e.g., up to M of the disk drives can fail) so that if any N of the $N+M$ encoded blocks are received successfully by the receiver, the original data (as well as the check data) can be reconstructed. $N/(N+M)$ is thus the code rate of the erasure code encoding (i.e., how much space the original data takes up in the encoded data). Erasure codes for select values of N and M can be implemented on RAID systems employing $N+M$ (disk) drives by spreading the original data among N “data” drives, and using the remaining M drives as “check” drives. Then, when any N of the $N+M$ drives are correctly functioning, the original data can be reconstructed, and the check data can be regenerated.

Erasure codes (or more specifically, erasure coding systems) are generally regarded as impractical for values of M larger than 1 (e.g., RAID5 systems, such as parity drive systems) or 2 (RAID6 systems), that is, for more than one or two check drives. For example, see H. Peter Anvin, “The mathematics of RAID-6,” the entire content of which is incorporated herein by reference, p. 7, “Thus, in 2-disk-degraded mode, performance will be very slow. However, it is expected

2

that that will be a rare occurrence, and that performance will not matter significantly in that case.” See also Robert Maddock et al., “Surviving Two Disk Failures,” p. 6, “The main difficulty with this technique is that calculating the check codes, and reconstructing data after failures, is quite complex. It involves polynomials and thus multiplication, and requires special hardware, or at least a signal processor, to do it at sufficient speed.” In addition, see also James S. Plank, “All About Erasure Codes: —Reed-Solomon Coding—LDPC Coding,” slide 15 (describing computational complexity of Reed-Solomon decoding), “Bottom line: When n & m grow, it is brutally expensive.” Accordingly, there appears to be a general consensus among experts in the field that erasure coding systems are impractical for RAID systems for all but small values of M (that is, small numbers of check drives), such as 1 or 2.

Modern disk drives, on the other hand, are much less reliable than those envisioned when RAID was proposed. This is due to their capacity growing out of proportion to their reliability. Accordingly, systems with only a single check disk have, for the most part, been discontinued in favor of systems with two check disks.

In terms of reliability, a higher check disk count is clearly more desirable than a lower check disk count. If the count of error events on different drives is larger than the check disk count, data may be lost and that cannot be reconstructed from the correctly functioning drives. Error events extend well beyond the traditional measure of advertised mean time between failures (MTBF). A simple, real world example is a service event on a RAID system where the operator mistakenly replaces the wrong drive or, worse yet, replaces a good drive with a broken drive. In the absence of any generally accepted methodology to train, certify, and measure the effectiveness of service technicians, these types of events occur at an unknown rate, but certainly occur. The foolproof solution for protecting data in the face of multiple error events is to increase the check disk count.

SUMMARY

Aspects of embodiments of the present invention address these problems by providing a practical erasure coding system that, for byte-level RAID processing (where each byte is made up of 8 bits), performs well even for values of $N+M$ as large as 256 drives (for example, $N=127$ data drives and $M=129$ check drives). Further aspects provide for a single precomputed encoding matrix (or master encoding matrix) S of size $M_{max} \times N_{max}$ or $(N_{max}+M_{max}) \times N_{max}$ or $(M_{max}-1) \times N_{max}$ elements (e.g., bytes), which can be used, for example, for any combination of $N \leq N_{max}$ data drives and $M \leq M_{max}$ check drives such that $N_{max}+M_{max} \leq 256$ (e.g., $N_{max}=127$ and $M_{max}=129$, or $N_{max}=63$ and $M_{max}=193$). This is an improvement over prior art solutions that rebuild such matrices from scratch every time N or M changes (such as adding another check drive). Still higher values of N and M are possible with larger processing increments, such as 2 bytes, which affords up to $N+M=65,536$ drives (such as $N=32,767$ data drives and $M=32,769$ check drives).

Higher check disk count can offer increased reliability and decreased cost. The higher reliability comes from factors such as the ability to withstand more drive failures. The decreased cost arises from factors such as the ability to create larger groups of data drives. For example, systems with two checks disks are typically limited to group sizes of 10 or fewer drives for reliability reasons. With a higher check disk count,

US 9,160,374 B2

3

larger groups are available, which can lead to fewer overall components for the same unit of storage and hence, lower cost.

Additional aspects of embodiments of the present invention further address these problems by providing a standard parity drive as part of the encoding matrix. For instance, aspects provide for a parity drive for configurations with up to 127 data drives and up to 128 (non-parity) check drives, for a total of up to 256 total drives including the parity drive. Further aspects provide for different breakdowns, such as up to 63 data drives, a parity drive, and up to 192 (non-parity) check drives. Providing a parity drive offers performance comparable to RAIDS in comparable circumstances (such as single data drive failures) while also being able to tolerate significantly larger numbers of data drive failures by including additional (non-parity) check drives.

Further aspects are directed to a system and method for implementing a fast solution matrix algorithm for Reed-Solomon codes. While known solution matrix algorithms compute an $N \times N$ solution matrix (see, for example, J. S. Plank, "A tutorial on Reed-Solomon coding for fault-tolerance in RAID-like systems," *Software—Practice & Experience*, 27(9):995-1012, September 1997, and J. S. Plank and Y. Ding, "Note: Correction to the 1997 tutorial on Reed-Solomon coding," Technical Report CS-03-504, University of Tennessee, April 2003), requiring $O(N^3)$ operations, regardless of the number of failed data drives, aspects of embodiments of the present invention compute only an $F \times F$ solution matrix, where F is the number of failed data drives. The overhead for computing this $F \times F$ solution matrix is approximately $F^3/3$ multiplication operations and the same number of addition operations. Not only is $F \leq N$, in almost any practical application, the number of failed data drives F is considerably smaller than the number of data drives N . Accordingly, the fast solution matrix algorithm is considerably faster than any known approach for practical values of F and N .

Still further aspects are directed toward fast implementations of the check data generation and the lost (original and check) data reconstruction. Some of these aspects are directed toward fetching the surviving (original and check) data a minimum number of times (that is, at most once) to carry out the data reconstruction. Some of these aspects are directed toward efficient implementations that can maximize or significantly leverage the available parallel processing power of multiple cores working concurrently on the check data generation and the lost data reconstruction. Existing implementations do not attempt to accelerate these aspects of the data generation and thus fail to achieve a comparable level of performance.

In an exemplary embodiment of the present invention, a system for accelerated error-correcting code (ECC) processing is provided. The system includes a processing core for executing computer instructions and accessing data from a main memory; and a non-volatile storage medium (for example, a disk drive, or flash memory) for storing the computer instructions. The processing core, the storage medium, and the computer instructions are configured to implement an erasure coding system. The erasure coding system includes a data matrix for holding original data in the main memory, a check matrix for holding check data in the main memory, an encoding matrix for holding first factors in the main memory, and a thread for executing on the processing core. The first factors are for encoding the original data into the check data. The thread includes a parallel multiplier for concurrently multiplying multiple data entries of a matrix by a single factor; and a first sequencer for ordering operations through

4

the data matrix and the encoding matrix using the parallel multiplier to generate the check data.

The first sequencer may be configured to access each entry of the data matrix from the main memory at most once while generating the check data.

The processing core may include a plurality of processing cores. The thread may include a plurality of threads. The erasure coding system may further include a scheduler for generating the check data by dividing the data matrix into a plurality of data matrices, dividing the check matrix into a plurality of check matrices, assigning corresponding ones of the data matrices and the check matrices to the threads, and assigning the threads to the processing cores to concurrently generate portions of the check data corresponding to the check matrices from respective ones of the data matrices.

The data matrix may include a first number of rows. The check matrix may include a second number of rows. The encoding matrix may include the second number of rows and the first number of columns.

The data matrix may be configured to add rows to the first number of rows or the check matrix may be configured to add rows to the second number of rows while the first factors remain unchanged.

Each of entries of one of the rows of the encoding matrix may include a multiplicative identity factor (such as 1).

The data matrix may be configured to be divided by rows into a surviving data matrix for holding surviving original data of the original data, and a lost data matrix corresponding to lost original data of the original data and including a third number of rows. The erasure coding system may further include a solution matrix for holding second factors in the main memory. The second factors are for decoding the check data into the lost original data using the surviving original data and the first factors.

The solution matrix may include the third number of rows and the third number of columns.

The solution matrix may further include an inverted said third number by said third number sub-matrix of the encoding matrix.

The erasure coding system may further include a first list of rows of the data matrix corresponding to the surviving data matrix, and a second list of rows of the data matrix corresponding to the lost data matrix.

The data matrix may be configured to be divided into a surviving data matrix for holding surviving original data of the original data, and a lost data matrix corresponding to lost original data of the original data. The erasure coding system may further include a solution matrix for holding second factors in the main memory. The second factors are for decoding the check data into the lost original data using the surviving original data and the first factors. The thread may further include a second sequencer for ordering operations through the surviving data matrix, the encoding matrix, the check matrix, and the solution matrix using the parallel multiplier to reconstruct the lost original data.

The second sequencer may be further configured to access each entry of the surviving data matrix from the main memory at most once while reconstructing the lost original data.

The processing core may include a plurality of processing cores. The thread may include a plurality of threads. The erasure coding system may further include: a scheduler for generating the check data and reconstructing the lost original data by dividing the data matrix into a plurality of data matrices; dividing the surviving data matrix into a plurality of surviving data matrices; dividing the lost data matrix into a plurality of lost data matrices; dividing the check matrix into a plurality of check matrices; assigning corresponding ones

US 9,160,374 B2

5

of the data matrices, the surviving data matrices, the lost data matrices, and the check matrices to the threads; and assigning the threads to the processing cores to concurrently generate portions of the check data corresponding to the check matrices from respective ones of the data matrices and to concurrently reconstruct portions of the lost original data corresponding to the lost data matrices from respective ones of the surviving data matrices and the check matrices.

The check matrix may be configured to be divided into a surviving check matrix for holding surviving check data of the check data, and a lost check matrix corresponding to lost check data of the check data. The second sequencer may be configured to order operations through the surviving data matrix, the reconstructed lost original data, and the encoding matrix using the parallel multiplier to regenerate the lost check data.

The second sequencer may be further configured to reconstruct the lost original data concurrently with regenerating the lost check data.

The second sequencer may be further configured to access each entry of the surviving data matrix from the main memory at most once while reconstructing the lost original data and regenerating the lost check data.

The second sequencer may be further configured to regenerate the lost check data without accessing the reconstructed lost original data from the main memory.

The processing core may include a plurality of processing cores. The thread may include a plurality of threads. The erasure coding system may further include a scheduler for generating the check data, reconstructing the lost original data, and regenerating the lost check data by: dividing the data matrix into a plurality of data matrices; dividing the surviving data matrix into a plurality of surviving data matrices; dividing the lost data matrix into a plurality of lost data matrices; dividing the check matrix into a plurality of check matrices; dividing the surviving check matrix into a plurality of surviving check matrices; dividing the lost check matrix into a plurality of lost check matrices; assigning corresponding ones of the data matrices, the surviving data matrices, the lost data matrices, the check matrices, the surviving check matrices, and the lost check matrices to the threads; and assigning the threads to the processing cores to concurrently generate portions of the check data corresponding to the check matrices from respective ones of the data matrices, to concurrently reconstruct portions of the lost original data corresponding to the lost data matrices from respective ones of the surviving data matrices and the surviving check matrices, and to concurrently regenerate portions of the lost check data corresponding to the lost check matrices from respective ones of the surviving data matrices and respective portions of the reconstructed lost original data.

The processing core may include 16 data registers. Each of the data registers may include 16 bytes. The parallel multiplier may be configured to process the data in units of at least 64 bytes spread over at least four of the data registers at a time.

Consecutive instructions to process each of the units of the data may access separate ones of the data registers to permit concurrent execution of the consecutive instructions by the processing core.

The parallel multiplier may include two lookup tables for doing concurrent multiplication of 4-bit quantities across 16 byte-sized entries using the PSHUFB (Packed Shuffle Bytes) instruction.

The parallel multiplier may be further configured to receive an input operand in four of the data registers, and return with the input operand intact in the four of the data registers.

6

According to another exemplary embodiment of the present invention, a method of accelerated error-correcting code (ECC) processing on a computing system is provided. The computing system includes a non-volatile storage medium (such as a disk drive or flash memory), a processing core for accessing instructions and data from a main memory, and a computer program including a plurality of computer instructions for implementing an erasure coding system. The method includes: storing the computer program on the storage medium; executing the computer instructions on the processing core; arranging original data as a data matrix in the main memory; arranging first factors as an encoding matrix in the main memory, the first factors being for encoding the original data into check data, the check data being arranged as a check matrix in the main memory; and generating the check data using a parallel multiplier for concurrently multiplying multiple data entries of a matrix by a single factor. The generating of the check data includes ordering operations through the data matrix and the encoding matrix using the parallel multiplier.

The generating of the check data may include accessing each entry of the data matrix from the main memory at most once.

The processing core may include a plurality of processing cores. The executing of the computer instructions may include executing the computer instructions on the processing cores. The method may further include scheduling the generating of the check data by: dividing the data matrix into a plurality of data matrices; dividing the check matrix into a plurality of check matrices; and assigning corresponding ones of the data matrices and the check matrices to the processing cores to concurrently generate portions of the check data corresponding to the check matrices from respective ones of the data matrices.

The method may further include: dividing the data matrix into a surviving data matrix for holding surviving original data of the original data, and a lost data matrix corresponding to lost original data of the original data; arranging second factors as a solution matrix in the main memory, the second factors being for decoding the check data into the lost original data using the surviving original data and the first factors; and reconstructing the lost original data by ordering operations through the surviving data matrix, the encoding matrix, the check matrix, and the solution matrix using the parallel multiplier.

The reconstructing of the lost original data may include accessing each entry of the surviving data matrix from the main memory at most once.

The processing core may include a plurality of processing cores. The executing of the computer instructions may include executing the computer instructions on the processing cores. The method may further include scheduling the generating of the check data and the reconstructing of the lost original data by: dividing the data matrix into a plurality of data matrices; dividing the surviving data matrix into a plurality of surviving data matrices; dividing the lost data matrix into a plurality of lost data matrices; dividing the check matrix into a plurality of check matrices; and assigning corresponding ones of the data matrices, the surviving data matrices, the lost data matrices, and the check matrices to the processing cores to concurrently generate portions of the check data corresponding to the check matrices from respective ones of the data matrices and to concurrently reconstruct portions of the lost original data corresponding to the lost data matrices from respective ones of the surviving data matrices and the check matrices.

US 9,160,374 B2

7

The method may further include: dividing the check matrix into a surviving check matrix for holding surviving check data of the check data, and a lost check matrix corresponding to lost check data of the check data; and regenerating the lost check data by ordering operations through the surviving data matrix, the reconstructed lost original data, and the encoding matrix using the parallel multiplier.

The reconstructing of the lost original data may take place concurrently with the regenerating of the lost check data.

The reconstructing of the lost original data and the regenerating of the lost check data may include accessing each entry of the surviving data matrix from the main memory at most once.

The regenerating of the lost check data may take place without accessing the reconstructed lost original data from the main memory.

The processing core may include a plurality of processing cores. The executing of the computer instructions may include executing the computer instructions on the processing cores. The method may further include scheduling the generating of the check data, the reconstructing of the lost original data, and the regenerating of the lost check data by: dividing the data matrix into a plurality of data matrices; dividing the surviving data matrix into a plurality of surviving data matrices; dividing the lost data matrix into a plurality of lost data matrices; dividing the check matrix into a plurality of check matrices; dividing the surviving check matrix into a plurality of surviving check matrices; dividing the lost check matrix into a plurality of lost check matrices; and assigning corresponding ones of the data matrices, the surviving data matrices, the lost data matrices, the check matrices, the surviving check matrices, and the lost check matrices to the processing cores to concurrently generate portions of the check data corresponding to the check matrices from respective ones of the data matrices, to concurrently reconstruct portions of the lost original data corresponding to the lost data matrices from respective ones of the surviving data matrices and the surviving check matrices, and to concurrently regenerate portions of the lost check data corresponding to the lost check matrices from respective ones of the surviving data matrices and respective portions of the reconstructed lost original data.

According to yet another exemplary embodiment of the present invention, a non-transitory computer-readable storage medium (such as a disk drive, a compact disk (CD), a digital video disk (DVD), flash memory, a universal serial bus (USB) drive, etc.) containing a computer program including a plurality of computer instructions for performing accelerated error-correcting code (ECC) processing on a computing system is provided. The computing system includes a processing core for accessing instructions and data from a main memory. The computer instructions are configured to implement an erasure coding system when executed on the computing system by performing the steps of arranging original data as a data matrix in the main memory; arranging first factors as an encoding matrix in the main memory, the first factors being for encoding the original data into check data, the check data being arranged as a check matrix in the main memory; and generating the check data using a parallel multiplier for concurrently multiplying multiple data entries of a matrix by a single factor. The generating of the check data includes ordering operations through the data matrix and the encoding matrix using the parallel multiplier.

The generating of the check data may include accessing each entry of the data matrix from the main memory at most once.

8

The processing core may include a plurality of processing cores. The computer instructions may be further configured to perform the step of scheduling the generating of the check data by: dividing the data matrix into a plurality of data matrices; dividing the check matrix into a plurality of check matrices; and assigning corresponding ones of the data matrices and the check matrices to the processing cores to concurrently generate portions of the check data corresponding to the check matrices from respective ones of the data matrices.

The computer instructions may be further configured to perform the steps of: dividing the data matrix into a surviving data matrix for holding surviving original data of the original data, and a lost data matrix corresponding to lost original data of the original data; arranging second factors as a solution matrix in the main memory, the second factors being for decoding the check data into the lost original data using the surviving original data and the first factors; and reconstructing the lost original data by ordering operations through the surviving data matrix, the encoding matrix, the check matrix, and the solution matrix using the parallel multiplier.

The computer instructions may be further configured to perform the steps of: dividing the check matrix into a surviving check matrix for holding surviving check data of the check data, and a lost check matrix corresponding to lost check data of the check data; and regenerating the lost check data by ordering operations through the surviving data matrix, the reconstructed lost original data, and the encoding matrix using the parallel multiplier.

The reconstructing of the lost original data and the regenerating of the lost check data may include accessing each entry of the surviving data matrix from the main memory at most once.

The processing core may include a plurality of processing cores. The computer instructions may be further configured to perform the step of scheduling the generating of the check data, the reconstructing of the lost original data, and the regenerating of the lost check data by: dividing the data matrix into a plurality of data matrices; dividing the surviving data matrix into a plurality of surviving data matrices; dividing the lost data matrix into a plurality of lost data matrices; dividing the check matrix into a plurality of check matrices; dividing the surviving check matrix into a plurality of surviving check matrices; dividing the lost check matrix into a plurality of lost check matrices; and assigning corresponding ones of the data matrices, the surviving data matrices, the lost data matrices, the check matrices, the surviving check matrices, and the lost check matrices to the processing cores to concurrently generate portions of the check data corresponding to the check matrices from respective ones of the data matrices, to concurrently reconstruct portions of the lost original data corresponding to the lost data matrices from respective ones of the surviving data matrices and the surviving check matrices, and to concurrently regenerate portions of the lost check data corresponding to the lost check matrices from respective ones of the surviving data matrices and respective portions of the reconstructed lost original data.

By providing practical and efficient systems and methods for erasure coding systems (which for byte-level processing can support up to $N+M=256$ drives, such as $N=127$ data drives and $M=129$ check drives, including a parity drive), applications such as RAID systems that can tolerate far more failing drives than was thought to be possible or practical can be implemented with accelerated performance significantly better than any prior art solution.

BRIEF DESCRIPTION OF THE DRAWINGS

The accompanying drawings, together with the specification, illustrate exemplary embodiments of the present invention and, together with the description, serve to explain aspects and principles of the present invention.

FIG. 1 shows an exemplary stripe of original and check data according to an embodiment of the present invention.

FIG. 2 shows an exemplary method for reconstructing lost data after a failure of one or more drives according to an embodiment of the present invention.

FIG. 3 shows an exemplary method for performing a parallel lookup Galois field multiplication according to an embodiment of the present invention.

FIG. 4 shows an exemplary method for sequencing the parallel lookup multiplier to perform the check data generation according to an embodiment of the present invention.

FIGS. 5-7 show an exemplary method for sequencing the parallel lookup multiplier to perform the lost data reconstruction according to an embodiment of the present invention.

FIG. 8 illustrates a multi-core architecture system according to an embodiment of the present invention.

FIG. 9 shows an exemplary disk drive configuration according to an embodiment of the present invention.

DETAILED DESCRIPTION

Hereinafter, exemplary embodiments of the invention will be described in more detail with reference to the accompanying drawings. In the drawings, like reference numerals refer to like elements throughout.

While optimal erasure codes have many applications, for ease of description, they will be described in this application with respect to RAID applications, i.e., erasure coding systems for the storage and retrieval of digital data distributed across numerous storage devices (or drives), though the present application is not limited thereto. For further ease of description, the storage devices will be assumed to be disk drives, though the invention is not limited thereto. In RAID systems, the data (or original data) is broken up into stripes, each of which includes N uniformly sized blocks (data blocks), and the N blocks are written across N separate drives (the data drives), one block per data drive.

In addition, for ease of description, blocks will be assumed to be composed of L elements, each element having a fixed size, say 8 bits or one byte. An element, such as a byte, forms the fundamental unit of operation for the RAID processing, but the invention is just as applicable to other size elements, such as 16 bits (2 bytes). For simplification, unless otherwise indicated, elements will be assumed to be one byte in size throughout the description that follows, and the term "element(s)" and "byte(s)" will be used synonymously.

Conceptually, different stripes can distribute their data blocks across different combinations of drives, or have different block sizes or numbers of blocks, etc., but for simplification and ease of description and implementation, the described embodiments in the present application assume a consistent block size (L bytes) and distribution of blocks among the data drives between stripes. Further, all variables, such as the number of data drives N, will be assumed to be positive integers unless otherwise specified. In addition, since the N=1 case reduces to simple data mirroring (that is, copying the same data drive multiple times), it will also be assumed for simplicity that N≥2 throughout.

The N data blocks from each stripe are combined using arithmetic operations (to be described in more detail below) in M different ways to produce M blocks of check data (check

blocks), and the M check blocks written across M drives (the check drives) separate from the N data drives, one block per check drive. These combinations can take place, for example, when new (or changed) data is written to (or back to) disk. Accordingly, each of the N+M drives (data drives and check drives) stores a similar amount of data, namely one block for each stripe. As the processing of multiple stripes is conceptually similar to the processing of one stripe (only processing multiple blocks per drive instead of one), it will be further assumed for simplification that the data being stored or retrieved is only one stripe in size unless otherwise indicated. It will also be assumed that the block size L is sufficiently large that the data can be consistently divided across each block to produce subsets of the data that include respective portions of the blocks (for efficient concurrent processing by different processing units).

FIG. 1 shows an exemplary stripe 10 of original and check data according to an embodiment of the present invention.

Referring to FIG. 1, the stripe 10 can be thought of not only as the original N data blocks 20 that make up the original data, but also the corresponding M check blocks 30 generated from the original data (that is, the stripe 10 represents encoded data). Each of the N data blocks 20 is composed of L bytes 25 (labeled byte 1, byte 2, . . . , byte L), and each of the M check blocks 30 is composed of L bytes 35 (labeled similarly). In addition, check drive 1, byte 1, is a linear combination of data drive 1, byte 1; data drive 2, byte 1; . . . ; data drive N, byte 1. Likewise, check drive 1, byte 2, is generated from the same linear combination formula as check drive 1, byte 1, only using data drive 1, byte 2; data drive 2, byte 2; . . . ; data drive N, byte 2. In contrast, check drive 2, byte 1, uses a different linear combination formula than check drive 1, byte 1, but applies it to the same data, namely data drive 1, byte 1; data drive 2, byte 1; . . . ; data drive N, byte 1. In this fashion, each of the other check bytes 35 is a linear combination of the respective bytes of each of the N data drives 20 and using the corresponding linear combination formula for the particular check drive 30.

The stripe 10 in FIG. 1 can also be represented as a matrix C of encoded data. C has two sub-matrices, namely original data D on top and check data J on bottom. That is,

$$C = \begin{bmatrix} D \\ J \end{bmatrix} = \begin{bmatrix} D_{11} & D_{12} & \dots & D_{1L} \\ D_{21} & D_{22} & \dots & D_{2L} \\ \vdots & \vdots & \ddots & \vdots \\ D_{N1} & D_{N2} & \dots & D_{NL} \\ J_{11} & J_{12} & \dots & J_{1L} \\ J_{21} & J_{22} & \dots & J_{2L} \\ \vdots & \vdots & \ddots & \vdots \\ J_{M1} & J_{M2} & \dots & J_{ML} \end{bmatrix},$$

where D_{ij} =byte j from data drive i and J_{ij} =byte j from check drive i. Thus, the rows of encoded data C represent blocks, while the columns represent corresponding bytes of each of the drives.

Further, in case of a disk drive failure of one or more disks, the arithmetic operations are designed in such a fashion that for any stripe, the original data (and by extension, the check data) can be reconstructed from any combination of N data and check blocks from the corresponding N+M data and check blocks that comprise the stripe. Thus, RAID provides both parallel processing (reading and writing the data in stripes across multiple drives concurrently) and fault tolerance (regeneration of the original data even if as many as M of

US 9,160,374 B2

11

the drives fail), at the computational cost of generating the check data any time new data is written to disk, or changed data is written back to disk, as well as the computational cost of reconstructing any lost original data and regenerating any lost check data after a disk failure.

For example, for M=1 check drive, a single parity drive can function as the check drive (i.e., a RAID4 system). Here, the arithmetic operation is bitwise exclusive OR of each of the N corresponding data bytes in each data block of the stripe. In addition, as mentioned earlier, the assignment of parity blocks from different stripes to the same drive (i.e., RAID4) or different drives (i.e., RAID5) is arbitrary, but it does simplify the description and implementation to use a consistent assignment between stripes, so that will be assumed throughout. Since M=1 reduces to the case of a single parity drive, it will further be assumed for simplicity that M≥2 throughout.

For such larger values of M, Galois field arithmetic is used to manipulate the data, as described in more detail later. Galois field arithmetic, for Galois fields of powers-of-2 (such as 2^P) numbers of elements, includes two fundamental operations: (1) addition (which is just bitwise exclusive OR, as with the parity drive-only operations for M=1), and (2) multiplication. While Galois field (GF) addition is trivial on standard processors, GF multiplication is not. Accordingly, a significant component of RAID performance for M≥2 is speeding up the performance of GF multiplication, as will be discussed later. For purposes of description, GF addition will be represented by the symbol + throughout while GF multiplication will be represented by the symbol × throughout.

Briefly, in exemplary embodiments of the present invention, each of the M check drives holds linear combinations (over GF arithmetic) of the N data drives of original data, one linear combination (i.e., a GF sum of N terms, where each term represents a byte of original data times a corresponding factor (using GF multiplication) for the respective data drive) for each check drive, as applied to respective bytes in each block. One such linear combination can be a simple parity, i.e., entirely GF addition (all factors equal 1), such as a GF sum of the first byte in each block of original data as described above.

The remaining M-1 linear combinations include more involved calculations that include the nontrivial GF multiplication operations (e.g., performing a GF multiplication of the first byte in each block by a corresponding factor for the respective data drive, and then performing a GF sum of all these products). These linear combinations can be represented by an (N+M)×N matrix (encoding matrix or information dispersal matrix (IDM)) E of the different factors, one factor for each combination of (data or check) drive and data drive, with one row for each of the N+M data and check drives and one column for each of the N data drives. The IDM E can also be represented as

$$\begin{bmatrix} I_N \\ H \end{bmatrix},$$

where I_N represents the N×N identity matrix (i.e., the original (unencoded) data) and H represents the M×N matrix of factors for the check drives (where each of the M rows corresponds to one of the M check drives and each of the N columns corresponds to one of the N data drives).

12

Thus,

$$E = \begin{bmatrix} I_N \\ H \end{bmatrix} = \begin{bmatrix} 1 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 1 \\ H_{11} & H_{12} & \dots & H_{1N} \\ H_{21} & H_{22} & \dots & H_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ H_{M1} & H_{M2} & \dots & H_{MN} \end{bmatrix},$$

where H_{ij}=factor for check drive i and data drive j. Thus, the rows of encoded data C represent blocks, while the columns represent corresponding bytes of each of the drives. In addition, check factors H, original data D, and check data J are related by the formula J=H×D (that is, matrix multiplication), or

$$\begin{bmatrix} J_{11} & J_{12} & \dots & J_{1L} \\ J_{21} & J_{22} & \dots & J_{2L} \\ \vdots & \vdots & \ddots & \vdots \\ J_{M1} & J_{M2} & \dots & J_{ML} \end{bmatrix} = \begin{bmatrix} H_{11} & H_{12} & \dots & H_{1N} \\ H_{21} & H_{22} & \dots & H_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ H_{M1} & H_{M2} & \dots & H_{MN} \end{bmatrix} \times \begin{bmatrix} D_{11} & D_{12} & \dots & D_{1L} \\ D_{21} & D_{22} & \dots & D_{2L} \\ \vdots & \vdots & \ddots & \vdots \\ D_{N1} & D_{N2} & \dots & D_{NL} \end{bmatrix},$$

where J₁₁=(H₁₁×D₁₁)+(H₁₂×D₂₁)+ . . . +W_{1,N}×D_{N1}), J₁₂=(H₁₁×D₁₂)+(H₁₂×D₂₂)+ . . . +W_{1,N}×D_{N2}), J₂₁=(H₂₁×D₁₁)+(H₂₂×D₂₁)+ . . . +(H_{2,N}×D_{N1}), and in general, J_{ij}=(H_{i1}×D_{1j})+(H_{i2}×D_{2j})+ . . . +(H_{i,N}×D_{Nj}) for 1≤i≤M and 1≤j≤L.

Such an encoding matrix E is also referred to as an information dispersal matrix (IDM). It should be noted that matrices such as check drive encoding matrix H and identity matrix I_N also represent encoding matrices, in that they represent matrices of factors to produce linear combinations over GF arithmetic of the original data. In practice, the identity matrix I_N is trivial and may not need to be constructed as part of the IDM E. Only the encoding matrix E, however, will be referred to as the IDM. Methods of building an encoding matrix such as IDM E or check drive encoding matrix H are discussed below. In further embodiments of the present invention (as discussed further in Appendix A), such (N+M)×N (or M×N) matrices can be trivially constructed (or simply indexed) from a master encoding matrix S, which is composed of (N_{max}+M_{max})×N_{max} (or M_{max}×N_{max}) bytes or elements, where N_{max}+M_{max}=256 (or some other power of two) and N≤N_{max} and M≤M_{max}. For example, one such master encoding matrix S can include a 127×127 element identity matrix on top (for up to N_{max}=127 data drives), a row of 1's (for a parity drive), and a 128×127 element encoding matrix on bottom (for up to M_{max}=129 check drives, including the parity drive), for a total of N_{max}+M_{max}=256 drives.

The original data, in turn, can be represented by an N×L matrix D of bytes, each of the N rows representing the L bytes of a block of the corresponding one of the N data drives. If C represents the corresponding (N+M)×L matrix of encoded bytes (where each of the N+M rows corresponds to one of the N+M data and check drives), then C can be represented as

13

$$E \times D = \begin{bmatrix} I_N \\ H \end{bmatrix} \times D = \begin{bmatrix} I_N \times D \\ H \times D \end{bmatrix} = \begin{bmatrix} D \\ J \end{bmatrix},$$

where $J=H \times D$ is an $M \times L$ matrix of check data, with each of the M rows representing the L check bytes of the corresponding one of the M check drives. It should be noted that in the relationships such as $C=E \times D$ or $J=H \times D$, \times represents matrix multiplication over the Galois field (i.e., GF multiplication and GF addition being used to generate each of the entries in, for example, C or J).

In exemplary embodiments of the present invention, the first row of the check drive encoding matrix H (or the $(N+1)^{th}$ row of the IDM E) can be all 1's, representing the parity drive. For linear combinations involving this row, the GF multiplication can be bypassed and replaced with a GF sum of the corresponding bytes since the products are all trivial products involving the identity element 1. Accordingly, in parity drive implementations, the check drive encoding matrix H can also be thought of as an $(M-1) \times N$ matrix of non-trivial factors (that is, factors intended to be used in GF multiplication and not just GF addition).

Much of the RAID processing involves generating the check data when new or changed data is written to (or back to) disk. The other significant event for RAID processing is when one or more of the drives fail (data or check drives), or for whatever reason become unavailable. Assume that in such a failure scenario, F data drives fail and G check drives fail, where F and G are nonnegative integers. If $F=0$, then only check drives failed and all of the original data D survived. In this case, the lost check data can be regenerated from the original data D .

Accordingly, assume at least one data drive fails, that is, $F \geq 1$, and let $K=N-F$ represent the number of data drives that survive. K is also a nonnegative integer. In addition, let X represent the surviving original data and Y represent the lost original data. That is, X is a $K \times L$ matrix composed of the K rows of the original data matrix D corresponding to the K surviving data drives, while Y is an $F \times L$ matrix composed of the F rows of the original data matrix D corresponding to the F failed data drives.

$$\begin{bmatrix} X \\ Y \end{bmatrix}$$

thus represents a permuted original data matrix D' (that is, the original data matrix D , only with the surviving original data X on top and the lost original data Y on bottom. It should be noted that once the lost original data Y is reconstructed, it can be combined with the surviving original data X to restore the original data D , from which the check data for any of the failed check drives can be regenerated.

It should also be noted that $M-G$ check drives survive. In order to reconstruct the lost original data Y , enough (that is, at least N) total drives must survive. Given that $K=N-F$ data drives survive, and that $M-G$ check drives survive, it follows that $(N-F)+(M-G) \geq N$ must be true to reconstruct the lost original data Y . This is equivalent to $F+G \leq M$ (i.e., no more than $F+G$ drives fail), or $F \leq M-G$ (that is, the number of failed data drives does not exceed the number of surviving check drives). It will therefore be assumed for simplicity that $F \leq M-G$.

In the routines that follow, performance can be enhanced by prebuilding lists of the failed and surviving data and check

14

drives (that is, four separate lists). This allows processing of the different sets of surviving and failed drives to be done more efficiently than existing solutions, which use, for example, bit vectors that have to be examined one bit at a time and often include large numbers of consecutive zeros (or ones) when ones (or zeros) are the bit values of interest.

FIG. 2 shows an exemplary method 300 for reconstructing lost data after a failure of one or more drives according to an embodiment of the present invention.

While the recovery process is described in more detail later, briefly it consists of two parts: (1) determining the solution matrix, and (2) reconstructing the lost data from the surviving data. Determining the solution matrix can be done in three steps with the following algorithm (Algorithm 1), with reference to FIG. 2:

1. (Step 310 in FIG. 2) Reducing the $(M+N) \times N$ IDM E to an $N \times N$ reduced encoding matrix T (also referred to as the transformed IDM) including the K surviving data drive rows and any F of the $M-G$ surviving check drive rows (for instance, the first F surviving check drive rows, as these will include the parity drive if it survived; recall that $F \leq M-G$ was assumed). In addition, the columns of the reduced encoding matrix T are rearranged so that the K columns corresponding to the K surviving data drives are on the left side of the matrix and the F columns corresponding to the F failed drives are on the right side of the matrix. (Step 320) These F surviving check drives selected to rebuild the lost original data Y will henceforth be referred to as "the F surviving check drives," and their check data W will be referred to as "the surviving check data," even though $M-G$ check drives survived. It should be noted that W is an $F \times L$ matrix composed of the F rows of the check data J corresponding to the F surviving check drives. Further, the surviving encoded data can be represented as a sub-matrix C' of the encoded data C . The surviving encoded data C' is an $N \times L$ matrix composed of the surviving original data X on top and the surviving check data W on bottom, that is,

$$C' = \begin{bmatrix} X \\ W \end{bmatrix}.$$

2. (Step 330) Splitting the reduced encoding matrix T into four sub-matrices (that are also encoding matrices): (i) a $K \times K$ identity matrix I_K (corresponding to the K surviving data drives) in the upper left, (ii) a $K \times F$ matrix O of zeros in the upper right, (iii) an $F \times K$ encoding matrix A in the lower left corresponding to the F surviving check drive rows and the K surviving data drive columns, and (iv) an $F \times F$ encoding matrix B in the lower right corresponding to the F surviving check drive rows and the F failed data drive columns. Thus, the reduced encoding matrix T can be represented as

$$\begin{bmatrix} I_K & O \\ A & B \end{bmatrix}.$$

3. (Step 340) Calculating the inverse B^{-1} of the $F \times F$ encoding matrix B . As is shown in more detail in Appendix A, $C'=T \times D'$, or

$$\begin{bmatrix} X \\ W \end{bmatrix} = \begin{bmatrix} I_K & O \\ A & B \end{bmatrix} \times \begin{bmatrix} X \\ Y \end{bmatrix}.$$

which is mathematically equivalent to $W=A \times X+B \times Y$. B^{-1} is the solution matrix, and is itself an $F \times F$ encoding matrix. Calculating the solution matrix B^{-1} thus allows the lost original data Y to be reconstructed from the encoding matrices A and B along with the surviving original data X and the surviving check data W .

The $F \times K$ encoding matrix A represents the original encoding matrix E , only limited to the K surviving data drives and the F surviving check drives. That is, each of the F rows of A represents a different one of the F surviving check drives, while each of the K columns of A represents a different one of the K surviving data drives. Thus, A provides the encoding factors needed to encode the original data for the surviving check drives, but only applied to the surviving data drives (that is, the surviving partial check data). Since the surviving original data X is available, A can be used to generate this surviving partial check data.

In similar fashion, the $F \times F$ encoding matrix B represents the original encoding matrix E , only limited to the F surviving check drives and the F failed data drives. That is, the F rows of B correspond to the same F rows of A , while each of the F columns of B represents a different one of the F failed data drives. Thus, B provides the encoding factors needed to encode the original data for the surviving check drives, but only applied to the failed data drives (that is, the lost partial check data). Since the lost original data Y is not available, B cannot be used to generate any of the lost partial check data. However, this lost partial check data can be determined from A and the surviving check data W . Since this lost partial check data represents the result of applying B to the lost original data Y , B^{-1} thus represents the necessary factors to reconstruct the lost original data Y from the lost partial check data.

It should be noted that steps 1 and 2 in Algorithm 1 above are logical, in that encoding matrices A and B (or the reduced encoding matrix T , for that matter) do not have to actually be constructed. Appropriate indexing of the IDM E (or the master encoding matrix S) can be used to obtain any of their entries. Step 3, however, is a matrix inversion over GF arithmetic and takes $O(F^3)$ operations, as discussed in more detail later. Nonetheless, this is a significant improvement over existing solutions, which require $O(N^3)$ operations, since the number of failed data drives F is usually significantly less than the number of data drives N in any practical situation.

(Step 350 in FIG. 2) Once the encoding matrix A and the solution matrix B^{-1} are known, reconstructing the lost data from the surviving data (that is, the surviving original data X and the surviving check data W) can be accomplished in four steps using the following algorithm (Algorithm 2):

1. Use A and the surviving original data X (using matrix multiplication) to generate the surviving check data (i.e., $A \times X$), only limited to the K surviving data drives. Call this limited check data the surviving partial check data.

2. Subtract this surviving partial check data from the surviving check data W (using matrix subtraction, i.e., $W-A \times X$, which is just entry-by-entry GF subtraction, which is the same as GF addition for this Galois field). This generates the surviving check data, only this time limited to the F failed data drives. Call this limited check data the lost partial check data.

3. Use the solution matrix B^{-1} and the lost partial check data (using matrix multiplication, i.e., $B^{-1} \times (W-A \times X)$) to reconstruct the lost original data Y . Call this the recovered original data Y .

4. Use the corresponding rows of the IDM E (or master encoding matrix S) for each of the G failed check drives along with the original data D , as reconstructed from the surviving and recovered original data X and Y , to regenerate the lost check data (using matrix multiplication).

As will be shown in more detail later, steps 1-3 together require $O(F)$ operations times the amount of original data D to reconstruct the lost original data Y for the F failed data drives (i.e., roughly 1 operation per failed data drive per byte of original data D), which is proportionally equivalent to the $O(M)$ operations times the amount of original data D needed to generate the check data J for the M check drives (i.e., roughly 1 operation per check drive per byte of original data D). In addition, this same equivalence extends to step 4, which takes $O(G)$ operations times the amount of original data D needed to regenerate the lost check data for the G failed check drives (i.e., roughly 1 operation per failed check drive per byte of original data D). In summary, the number of operations needed to reconstruct the lost data is $O(F+G)$ times the amount of original data D (i.e., roughly 1 operation per failed drive (data or check) per byte of original data D). Since $F+G \leq M$, this means that the computational complexity of Algorithm 2 (reconstructing the lost data from the surviving data) is no more than that of generating the check data J from the original data D .

As mentioned above, for exemplary purposes and ease of description, data is assumed to be organized in 8-bit bytes, each byte capable of taking on $2^8=256$ possible values. Such data can be manipulated in byte-size elements using GF arithmetic for a Galois field of size $2^8=256$ elements. It should also be noted that the same mathematical principles apply to any power-of-two Z number of elements, not just 256, as Galois fields can be constructed for any integral power of a prime number. Since Galois fields are finite, and since GF operations never overflow, all results are the same size as the inputs, for example, 8 bits.

In a Galois field of a power-of-two number of elements, addition and subtraction are the same operation, namely a bitwise exclusive OR (XOR) of the two operands. This is a very fast operation to perform on any current processor. It can also be performed on multiple bytes concurrently. Since the addition and subtraction operations take place, for example, on a byte-level basis, they can be done in parallel by using, for instance, x86 architecture Streaming SIMD Extensions (SSE) instructions (SIMD stands for single instruction, multiple data, and refers to performing the same instruction on different pieces of data, possibly concurrently), such as PXOR (Packed (bitwise) Exclusive OR).

SSE instructions can process, for example, 16-byte registers (XMM registers), and are able to process such registers as though they contain 16 separate one-byte operands (or 8 separate two-byte operands, or four separate four-byte operands, etc.) Accordingly, SSE instructions can do byte-level processing 16 times faster than when compared to processing a byte at a time. Further, there are 16 XMM registers, so dedicating four such registers for operand storage allows the data to be processed in 64-byte increments, using the other 12 registers for temporary storage. That is, individual operations can be performed as four consecutive SSE operations on the four respective registers (64 bytes), which can often allow such instructions to be efficiently pipelined and/or concurrently executed by the processor. In addition, the SSE instructions allows the same processing to be performed on different such 64-byte increments of data in parallel using different cores. Thus, using four separate cores can potentially speed up this processing by an additional factor of 4 over using a single core.

For example, a parallel adder (Parallel Adder) can be built using the 16-byte XMM registers and four consecutive PXOR instructions. Such parallel processing (that is, 64 bytes at a time with only a few machine-level instructions) for GF arithmetic is a significant improvement over doing the addition

17

one byte at a time. Since the data is organized in blocks of any fixed number of bytes, such as 4096 bytes (4 kilobytes, or 4 KB) or 32,768 bytes (32 KB), a block can be composed of numerous such 64-byte chunks (e.g., 64 separate 64-byte chunks in 4 KB, or 512 chunks in 32 KB).

Multiplication in a Galois field is not as straightforward. While much of it is bitwise shifts and exclusive OR's (i.e., "additions") that are very fast operations, the numbers "wrap" in peculiar ways when they are shifted outside of their normal bounds (because the field has only a finite set of elements), which can slow down the calculations. This "wrapping" in the GF multiplication can be addressed in many ways. For example, the multiplication can be implemented serially (Serial Multiplier) as a loop iterating over the bits of one operand while performing the shifts, adds, and wraps on the other operand. Such processing, however, takes several machine instructions per bit for 8 separate bits. In other words, this technique requires dozens of machine instructions per byte being multiplied. This is inefficient compared to, for example, the performance of the Parallel Adder described above.

For another approach (Serial Lookup Multiplier), multiplication tables (of all the possible products, or at least all the non-trivial products) can be pre-computed and built ahead of time. For example, a table of $256 \times 256 = 65,536$ bytes can hold all the possible products of the two different one-byte operands). However, such tables can force serialized access on what are only byte-level operations, and not take advantage of wide (concurrent) data paths available on modern processors, such as those used to implement the Parallel Adder above.

In still another approach (Parallel Multiplier), the GF multiplication can be done on multiple bytes at a time, since the same factor in the encoding matrix is multiplied with every element in a data block. Thus, the same factor can be multiplied with 64 consecutive data block bytes at a time. This is similar to the Parallel Adder described above, only there are several more operations needed to perform the operation. While this can be implemented as a loop on each bit of the factor, as described above, only performing the shifts, adds, and wraps on 64 bytes at a time, it can be more efficient to process the 256 possible factors as a (C language) switch statement, with inline code for each of 256 different combinations of two primitive GF operations: Multiply-by-2 and Add. For example, GF multiplication by the factor 3 can be effected by first doing a Multiply-by-2 followed by an Add. Likewise, GF multiplication by 4 is just a Multiply-by-2 followed by a Multiply-by-2 while multiplication by 6 is a Multiply-by-2 followed by an Add and then by another Multiply-by-2.

While this Add is identical to the Parallel Adder described above (e.g., four consecutive PXOR instructions to process 64 separate bytes), Multiply-by-2 is not as straightforward. For example, Multiply-by-2 in GF arithmetic can be implemented across 64 bytes at a time in 4 XMM registers via 4 consecutive PXOR instructions, 4 consecutive PCMPGTB (Packed Compare for Greater Than) instructions, 4 consecutive PADDB (Packed Add) instructions, 4 consecutive PAND (Bitwise AND) instructions, and 4 consecutive PXOR instructions. Though this takes 20 machine instructions, the instructions are very fast and results in 64 consecutive bytes of data at a time being multiplied by 2.

For 64 bytes of data, assuming a random factor between 0 and 255, the total overhead for the Parallel Multiplier is about 6 calls to multiply-by-2 and about 3.5 calls to add, or about $6 \times 20 + 3.5 \times 4 = 134$ machine instructions, or a little over 2 machine instructions per byte of data. While this compares favorably with byte-level processing, it is still possible to improve on this by building a parallel multiplier with a table

18

lookup (Parallel Lookup Multiplier) using the PSHUFB (Packed Shuffle Bytes) instruction and doing the GF multiplication in 4-bit nibbles (half bytes).

FIG. 3 shows an exemplary method 400 for performing a parallel lookup Galois field multiplication according to an embodiment of the present invention.

Referring to FIG. 3, in step 410, two lookup tables are built once: one lookup table for the low-order nibbles in each byte, and one lookup table for the high-order nibbles in each byte. Each lookup table contains 256 sets (one for each possible factor) of the 16 possible GF products of that factor and the 16 possible nibble values. Each lookup table is thus $256 \times 16 = 4096$ bytes, which is considerably smaller than the 65,536 bytes needed to store a complete one-byte multiplication table. In addition, PSHUFB does 16 separate table lookups at once, each for one nibble, so 8 PSHUFB instructions can be used to do all the table lookups for 64 bytes (128 nibbles).

Next, in step 420, the Parallel Lookup Multiplier is initialized for the next set of 64 bytes of operand data (such as original data or surviving original data). In order to save loading this data from memory on succeeding calls, the Parallel Lookup Multiplier dedicates four registers for this data, which are left intact upon exit of the Parallel Lookup Multiplier. This allows the same data to be called with different factors (such as processing the same data for another check drive).

Next in step 430, to process these 64 bytes of operand data, the Parallel Lookup Multiplier can be implemented with 2 MOVDQA (Move Double Quadword Aligned) instructions (from memory) to do the two table lookups and 4 MOVDQA instructions (register to register) to initialize registers (such as the output registers). These are followed in steps 440 and 450 by two nearly identical sets of 17 register-to-register instructions to carry out the multiplication 32 bytes at a time. Each such set starts (in step 440) with 5 more MOVDQA instructions for further initialization, followed by 2 PSRLW (Packed Shift Right Logical Word) instructions to realign the high-order nibbles for PSHUFB, and 4 PAND instructions to clear the high-order nibbles for PSHUFB. That is, two registers of byte operands are converted into four registers of nibble operands. Then, in step 450, 4 PSHUFB instructions are used to do the parallel table lookups, and 2 PXOR instructions to add the results of the multiplication on the two nibbles to the output registers.

Thus, the Parallel Lookup Multiplier uses 40 machine instructions to perform the parallel multiplication on 64 separate bytes, which is considerably better than the average 134 instructions for the Parallel Multiplier above, and only 10 times as many instructions as needed for the Parallel Adder. While some of the Parallel Lookup Multiplier's instructions are more complex than those of the Parallel Adder, much of this complexity can be concealed through the pipelined and/or concurrent execution of numerous such contiguous instructions (accessing different registers) on modern pipelined processors. For example, in exemplary implementations, the Parallel Lookup Multiplier has been timed at about 15 CPU clock cycles per 64 bytes processed per CPU core (about 0.36 clock cycles per instruction). In addition, the code footprint is practically nonexistent for the Parallel Lookup Multiplier (40 instructions) compared to that of the Parallel Multiplier (about 34,300 instructions), even when factoring the 8 KB needed for the two lookup tables in the Parallel Lookup Multiplier.

In addition, embodiments of the Parallel Lookup Multiplier can be passed 64 bytes of operand data (such as the next 64 bytes of surviving original data X to be processed) in four

consecutive registers, whose contents can be preserved upon exiting the Parallel Lookup Multiplier (and all in the same 40 machine instructions) such that the Parallel Lookup Multiplier can be invoked again on the same 64 bytes of data without having to access main memory to reload the data. Through such a protocol, memory accesses can be minimized (or significantly reduced) for accessing the original data D during check data generation or the surviving original data X during lost data reconstruction.

Further embodiments of the present invention are directed towards sequencing this parallel multiplication (and other GF) operations. While the Parallel Lookup Multiplier processes a GF multiplication of 64 bytes of contiguous data times a specified factor, the calls to the Parallel Lookup Multiplier should be appropriately sequenced to provide efficient processing. One such sequencer (Sequencer 1), for example, can generate the check data J from the original data D, and is described further with respect to FIG. 4.

The parity drive does not need GF multiplication. The check data for the parity drive can be obtained, for example, by adding corresponding 64-byte chunks for each of the data drives to perform the parity operation. The Parallel Adder can do this using 4 instructions for every 64 bytes of data for each of the N data drives, or N/16 instructions per byte.

The M-1 non-parity check drives can invoke the Parallel Lookup Multiplier on each 64-byte chunk, using the appropriate factor for the particular combination of data drive and check drive. One consideration is how to handle the data access. Two possible ways are:

- 1) "column-by-column," i.e., 64 bytes for one data drive, followed by the next 64 bytes for that data drive, etc., and adding the products to the running total in memory (using the Parallel Adder) before moving onto the next row (data drive); and
- 2) "row-by-row," i.e., 64 bytes for one data drive, followed by the corresponding 64 bytes for the next data drive, etc., and keeping a running total using the Parallel Adder, then moving onto the next set of 64-byte chunks.

Column-by-column can be thought of as "constant factor, varying data," in that the (GF multiplication) factor usually remains the same between iterations while the (64-byte) data changes with each iteration. Conversely, row-by-row can be thought of as "constant data, varying factor," in that the data usually remains the same between iterations while the factor changes with each iteration.

Another consideration is how to handle the check drives. Two possible ways are:

- a) one at a time, i.e., generate all the check data for one check drive before moving onto the next check drive; and
- b) all at once, i.e., for each 64-byte chunk of original data, do all of the processing for each of the check drives before moving onto the next chunk of original data.

While each of these techniques performs the same basic operations (e.g., 40 instructions for every 64 bytes of data for each of the N data drives and M-1 non-parity check drives, or $5N(M-1)/8$ instructions per byte for the Parallel Lookup Multiplier), empirical results show that combination (2)(b), that is, row-by-row data access on all of the check drives between data accesses performs best with the Parallel Lookup Multiplier. One reason may be that such an approach appears to minimize the number of memory accesses (namely, one) to each chunk of the original data D to generate the check data J. This embodiment of Sequencer 1 is described in more detail with reference to FIG. 4.

FIG. 4 shows an exemplary method 500 for sequencing the Parallel Lookup Multiplier to perform the check data generation according to an embodiment of the present invention.

Referring to FIG. 4, in step 510, the Sequencer 1 is called. Sequencer 1 is called to process multiple 64-byte chunks of data for each of the blocks across a stripe of data. For instance, Sequencer 1 could be called to process 512 bytes from each block. If, for example, the block size L is 4096 bytes, then it would take eight such calls to Sequencer 1 to process the entire stripe. The other such seven calls to Sequencer 1 could be to different processing cores, for instance, to carry out the check data generation in parallel. The number of 64-byte chunks to process at a time could depend on factors such as cache dimensions, input/output data structure sizes, etc.

In step 520, the outer loop processes the next 64-byte chunk of data for each of the drives. In order to minimize the number of accesses of each data drive's 64-byte chunk of data from memory, the data is loaded only once and preserved across calls to the Parallel Lookup Multiplier. The first data drive is handled specially since the check data has to be initialized for each check drive. Using the first data drive to initialize the check data saves doing the initialization as a separate step followed by updating it with the first data drive's data. In addition to the first data drive, the first check drive is also handled specially since it is a parity drive, so its check data can be initialized to the first data drive's data directly without needing the Parallel Lookup Multiplier.

In step 530, the first middle loop is called, in which the remainder of the check drives (that is, the non-parity check drives) have their check data initialized by the first data drive's data. In this case, there is a corresponding factor (that varies with each check drive) that needs to be multiplied with each of the first data drive's data bytes. This is handled by calling the Parallel Lookup Multiplier for each non-parity check drive.

In step 540, the second middle loop is called, which processes the other data drives' corresponding 64-byte chunks of data. As with the first data drive, each of the other data drives is processed separately, loading the respective 64 bytes of data into four registers (preserved across calls to the Parallel Lookup Multiplier). In addition, since the first check drive is the parity drive, its check data can be updated by directly adding these 64 bytes to it (using the Parallel Adder) before handling the non-parity check drives.

In step 550, the inner loop is called for the next data drive. In the inner loop (as with the first middle loop), each of the non-parity check drives is associated with a corresponding factor for the particular data drive. The factor is multiplied with each of the next data drive's data bytes using the Parallel Lookup Multiplier, and the results added to the check drive's check data.

Another such sequencer (Sequencer 2) can be used to reconstruct the lost data from the surviving data (using Algorithm 2). While the same column-by-column and row-by-row data access approaches are possible, as well as the same choices for handling the check drives, Algorithm 2 adds another dimension of complexity because of the four separate steps and whether to: (i) do the steps completely serially or (ii) do some of the steps concurrently on the same data. For example, step 1 (surviving check data generation) and step 4 (lost check data regeneration) can be done concurrently on the same data to reduce or minimize the number of surviving original data accesses from memory.

Empirical results show that method (2)(b)(ii), that is, row-by-row data access on all of the check drives and for both surviving check data generation and lost check data regeneration between data accesses performs best with the Parallel

Lookup Multiplier when reconstructing lost data using Algorithm 2. Again, this may be due to the apparent minimization of the number of memory accesses (namely, one) of each chunk of surviving original data X to reconstruct the lost data and the absence of memory accesses of reconstructed lost original data Y when regenerating the lost check data. This embodiment of Sequencer 1 is described in more detail with reference to FIGS. 5-7.

FIGS. 5-7 show an exemplary method 600 for sequencing the Parallel Lookup Multiplier to perform the lost data reconstruction according to an embodiment of the present invention.

Referring to FIG. 5, in step 610, the Sequencer 2 is called. Sequencer 2 has many similarities with the embodiment of Sequencer 1 illustrated in FIG. 4. For instance, Sequencer 2 processes the data drive data in 64-byte chunks like Sequencer 1. Sequencer 2 is more complex, however, in that only some of the data drive data is surviving; the rest has to be reconstructed. In addition, lost check data needs to be regenerated. Like Sequencer 1, Sequencer 2 does these operations in such a way as to minimize memory accesses of the data drive data (by loading the data once and calling the Parallel Lookup Multiplier multiple times). Assume for ease of description that there is at least one surviving data drive; the case of no surviving data drives is handled a little differently, but not significantly different. In addition, recall from above that the driving formula behind data reconstruction is $Y=B^{-1}\times(W-A\times X)$, where Y is the lost original data, B^{-1} is the solution matrix, W is the surviving check data, A is the partial check data encoding matrix (for the surviving check drives and the surviving data drives), and X is the surviving original data.

In step 620, the outer loop processes the next 64-byte chunk of data for each of the drives. Like Sequencer 1, the first surviving data drive is again handled specially since the partial check data $A\times X$ has to be initialized for each surviving check drive.

In step 630, the first middle loop is called, in which the partial check data $A\times X$ is initialized for each surviving check drive based on the first surviving data drive's 64 bytes of data. In this case, the Parallel Lookup Multiplier is called for each surviving check drive with the corresponding factor (from A) for the first surviving data drive.

In step 640, the second middle loop is called, in which the lost check data is initialized for each failed check drive. Using the same 64 bytes of the first surviving data drive (preserved across the calls to Parallel Lookup Multiplier in step 630), the Parallel Lookup Multiplier is again called, this time to initialize each of the failed check drive's check data to the corresponding component from the first surviving data drive. This completes the computations involving the first surviving data drive's 64 bytes of data, which were fetched with one access from main memory and preserved in the same four registers across steps 630 and 640.

Continuing with FIG. 6, in step 650, the third middle loop is called, which processes the other surviving data drives' corresponding 64-byte chunks of data. As with the first surviving data drive, each of the other surviving data drives is processed separately, loading the respective 64 bytes of data into four registers (preserved across calls to the Parallel Lookup Multiplier).

In step 660, the first inner loop is called, in which the partial check data $A\times X$ is updated for each surviving check drive based on the next surviving data drive's 64 bytes of data. In this case, the Parallel Lookup Multiplier is called for each surviving check drive with the corresponding factor (from A) for the next surviving data drive.

In step 670, the second inner loop is called, in which the lost check data is updated for each failed check drive. Using the same 64 bytes of the next surviving data drive (preserved across the calls to Parallel Lookup Multiplier in step 660), the Parallel Lookup Multiplier is again called, this time to update each of the failed check drive's check data by the corresponding component from the next surviving data drive. This completes the computations involving the next surviving data drive's 64 bytes of data, which were fetched with one access from main memory and preserved in the same four registers across steps 660 and 670.

Next, in step 680, the computation of the partial check data $A\times X$ is complete, so the surviving check data W is added to this result (recall that $W-A\times X$ is equivalent to $W+A\times X$ in binary Galois Field arithmetic). This is done by the fourth middle loop, which for each surviving check drive adds the corresponding 64-byte component of surviving check data W to the (surviving) partial check data $A\times X$ (using the Parallel Adder) to produce the (lost) partial check data $W-A\times X$.

Continuing with FIG. 7, in step 690, the fifth middle loop is called, which performs the two dimensional matrix multiplication $B^{-1}\times(W-A\times X)$ to produce the lost original data Y . The calculation is performed one row at a time, for a total of F rows, initializing the row to the first term of the corresponding linear combination of the solution matrix B^{-1} and the lost partial check data $W-A\times X$ (using the Parallel Lookup Multiplier).

In step 700, the third inner loop is called, which completes the remaining $F-1$ terms of the corresponding linear combination (using the Parallel Lookup Multiplier on each term) from the fifth middle loop in step 690 and updates the running calculation (using the Parallel Adder) of the next row of $B^{-1}\times(W-A\times X)$. This completes the next row (and reconstructs the corresponding failed data drive's lost data) of lost original data Y , which can then be stored at an appropriate location.

In step 710, the fourth inner loop is called, in which the lost check data is updated for each failed check drive by the newly reconstructed lost data for the next failed data drive. Using the same 64 bytes of the next reconstructed lost data (preserved across calls to the Parallel Lookup Multiplier), the Parallel Lookup Multiplier is called to update each of the failed check drives' check data by the corresponding component from the next failed data drive. This completes the computations involving the next failed data drive's 64 bytes of reconstructed data, which were performed as soon as the data was reconstructed and without being stored and retrieved from main memory.

Finally, in step 720, the sixth middle loop is called. The lost check data has been regenerated, so in this step, the newly regenerated check data is stored at an appropriate location (if desired).

Aspects of the present invention can be also realized in other environments, such as two-byte quantities, each such two-byte quantity capable of taking on $2^{16}=65,536$ possible values, by using similar constructs (scaled accordingly) to those presented here. Such extensions would be readily apparent to one of ordinary skill in the art, so their details will be omitted for brevity of description.

Exemplary techniques and methods for doing the Galois field manipulation and other mathematics behind RAID error correcting codes are described in Appendix A, which contains a paper "Information Dispersal Matrices for RAID Error Correcting Codes" prepared for the present application.

Multi-Core Considerations

What follows is an exemplary embodiment for optimizing or improving the performance of multi-core architecture sys-

tems when implementing the described erasure coding system routines. In multi-core architecture systems, each processor die is divided into multiple CPU cores, each with their own local caches, together with a memory (bus) interface and possible on-die cache to interface with a shared memory with other processor dies.

FIG. 8 illustrates a multi-core architecture system **100** having two processor dies **110** (namely, Die 0 and Die 1).

Referring to FIG. 8, each die **110** includes four central processing units (CPUs or cores) **120** each having a local level 1 (L1) cache. Each core **120** may have separate functional units, for example, an x86 execution unit (for traditional instructions) and a SSE execution unit (for software designed for the newer SSE instruction set). An example application of these function units is that the x86 execution unit can be used for the RAID control logic software while the SSE execution unit can be used for the GF operation software. Each die **110** also has a level 2 (L2) cache/memory bus interface **130** shared between the four cores **120**. Main memory **140**, in turn, is shared between the two dies **110**, and is connected to the input/output (I/O) controllers **150** that access external devices such as disk drives or other non-volatile storage devices via interfaces such as Peripheral Component Interconnect (PCI).

Redundant array of independent disks (RAID) controller processing can be described as a series of states or functions. These states may include: (1) Command Processing, to validate and schedule a host request (for example, to load or store data from disk storage); (2) Command Translation and Submission, to translate the host request into multiple disk requests and to pass the requests to the physical disks; (3) Error Correction, to generate check data and reconstruct lost data when some disks are not functioning correctly; and (4) Request Completion, to move data from internal buffers to requestor buffers. Note that the final state, Request Completion, may only be needed for a RAID controller that supports caching, and can be avoided in a cacheless design.

Parallelism is achieved in the embodiment of FIG. 8 by assigning different cores **120** to different tasks. For example, some of the cores **120** can be “command cores,” that is, assigned to the I/O operations, which includes reading and storing the data and check bytes to and from memory **140** and the disk drives via the I/O interface **150**. Others of the cores **120** can be “data cores,” and assigned to the GF operations, that is, generating the check data from the original data, reconstructing the lost data from the surviving data, etc., including the Parallel Lookup Multiplier and the sequencers described above. For example, in exemplary embodiments, a scheduler can be used to divide the original data **D** into corresponding portions of each block, which can then be processed independently by different cores **120** for applications such as check data generation and lost data reconstruction.

One of the benefits of this data core/command core subdivision of processing is ensuring that different code will be executed in different cores **120** (that is, command code in command cores, and data code in data cores). This improves the performance of the associated L1 cache in each core **120**, and avoids the “pollution” of these caches with code that is less frequently executed. In addition, empirical results show that the dies **110** perform best when only one core **120** on each die **110** does the GF operations (i.e., Sequencer 1 or Sequencer 2, with corresponding calls to Parallel Lookup Multiplier) and the other cores **120** do the I/O operations. This helps localize the Parallel Lookup Multiplier code and associated data to a single core **120** and not compete with other

cores **120**, while allowing the other cores **120** to keep the data moving between memory **140** and the disk drives via the I/O interface **150**.

Embodiments of the present invention yield scalable, high performance RAID systems capable of outperforming other systems, and at much lower cost, due to the use of high volume commodity components that are leveraged to achieve the result. This combination can be achieved by utilizing the mathematical techniques and code optimizations described elsewhere in this application with careful placement of the resulting code on specific processing cores. Embodiments can also be implemented on fewer resources, such as single-core dies and/or single-die systems, with decreased parallelism and performance optimization.

The process of subdividing and assigning individual cores **120** and/or dies **110** to inherently parallelizable tasks will result in a performance benefit. For example, on a Linux system, software may be organized into “threads,” and threads may be assigned to specific CPUs and memory systems via the `kthread_bind` function when the thread is created. Creating separate threads to process the GF arithmetic allows parallel computations to take place, which multiplies the performance of the system.

Further, creating multiple threads for command processing allows for fully overlapped execution of the command processing states. One way to accomplish this is to number each command, then use the arithmetic MOD function (`%` in C language) to choose a separate thread for each command. Another technique is to subdivide the data processing portion of each command into multiple components, and assign each component to a separate thread.

FIG. 9 shows an exemplary disk drive configuration **200** according to an embodiment of the present invention.

Referring to FIG. 9, eight disks are shown, though this number can vary in other embodiments. The disks are divided into three types: data drives **210**, parity drive **220**, and check drives **230**. The eight disks break down as three data drives **210**, one parity drive **220**, and four check drives **230** in the embodiment of FIG. 9.

Each of the data drives **210** is used to hold a portion of data. The data is distributed uniformly across the data drives **210** in stripes, such as 192 KB stripes. For example, the data for an application can be broken up into stripes of 192 KB, and each of the stripes in turn broken up into three 64 KB blocks, each of the three blocks being written to a different one of the three data drives **210**.

The parity drive **220** is a special type of check drive in that the encoding of its data is a simple summation (recall that this is exclusive OR in binary GF arithmetic) of the corresponding bytes of each of the three data drives **210**. That is, check data generation (Sequencer 1) or regeneration (Sequencer 2) can be performed for the parity drive **220** using the Parallel Adder (and not the Parallel Lookup Multiplier). Accordingly, the check data for the parity drive **220** is relatively straightforward to build. Likewise, when one of the data drives **210** no longer functions correctly, the parity drive **220** can be used to reconstruct the lost data by adding (same as subtracting in binary GF arithmetic) the corresponding bytes from each of the two remaining data drives **210**. Thus, a single drive failure of one of the data drives **210** is very straightforward to handle when the parity drive **220** is available (no Parallel Lookup Multiplier). Accordingly, the parity drive **220** can replace much of the GF multiplication operations with GF addition for both check data generation and lost data reconstruction.

Each of the check drives **230** contains a linear combination of the corresponding bytes of each of the data drives **210**. The linear combination is different for each check drive **230**, but in

general is represented by a summation of different multiples of each of the corresponding bytes of the data drives **210** (again, all arithmetic being GF arithmetic). For example, for the first check drive **230**, each of the bytes of the first data drive **210** could be multiplied by 4, each of the bytes of the second data drive **210** by 3, and each of the bytes of the third data drive **210** by 6, then the corresponding products for each of the corresponding bytes could be added to produce the first check drive data. Similar linear combinations could be used to produce the check drive data for the other check drives **230**. The specifics of which multiples for which check drive are explained in Appendix A.

With the addition of the parity drive **220** and check drives **230**, eight drives are used in the RAID system **200** of FIG. 9. Accordingly, each 192 KB of original data is stored as 512 KB (i.e., eight blocks of 64 KB) of (original plus check) data. Such a system **200**, however, is capable of recovering all of the original data provided any three of these eight drives survive. That is, the system **200** can withstand a concurrent failure of up to any five drives and still preserve all of the original data.

Exemplary Routines to Implement an Embodiment

The error correcting code (ECC) portion of an exemplary embodiment of the present invention may be written in software as, for example, four functions, which could be named as ECCInitialize, ECCSolve, ECCGenerate, and ECCRegenerate. The main functions that perform work are ECCGenerate and ECCRegenerate. ECCGenerate generates check codes for data that are used to recover data when a drive suffers an outage (that is, ECCGenerate generates the check data J from the original data D using Sequencer 1). ECCRegenerate uses these check codes and the remaining data to recover data after such an outage (that is, ECCRegenerate uses the surviving check data W, the surviving original data X, and Sequencer 2 to reconstruct the lost original data Y while also regenerating any of the lost check data). Prior to calling either of these functions, ECCSolve is called to compute the constants used for a particular configuration of data drives, check drives, and failed drives (for example, ECCSolve builds the solution matrix B^{-1} together with the lists of surviving and failed data and check drives). Prior to calling ECCSolve, ECCInitialize is called to generate constant tables used by all of the other functions (for example, ECCInitialize builds the IDM E and the two lookup tables for the Parallel Lookup Multiplier).

ECCInitialize

The function ECCInitialize creates constant tables that are used by all subsequent functions. It is called once at program initialization time. By copying or precomputing these values up front, these constant tables can be used to replace more time-consuming operations with simple table look-ups (such as for the Parallel Lookup Multiplier). For example, four tables useful for speeding up the GF arithmetic include:

1. mvct—an array of constants used to perform GF multiplication with the PSHUFQ instruction that operates on SSE registers (that is, the Parallel Lookup Multiplier).

2. mast—contains the master encoding matrix S (or the Information Dispersal Matrix (IDM) E, as described in Appendix A), or at least the nontrivial portion, such as the check drive encoding matrix H

3. mul_tab—contains the results of all possible GF multiplication operations of any two operands (for example, $256 \times 256 = 65,536$ bytes for all of the possible products of two different one-byte quantities)

4. div_tab—contains the results of all possible GF division operations of any two operands (can be similar in size to mul_tab)

ECCSolve

The function ECCSolve creates constant tables that are used to compute a solution for a particular configuration of data drives, check drives, and failed drives. It is called prior to using the functions ECCGenerate or ECCRegenerate. It allows the user to identify a particular case of failure by describing the logical configuration of data drives, check drives, and failed drives. It returns the constants, tables, and lists used to either generate check codes or regenerate data. For example, it can return the matrix B that needs to be inverted as well as the inverted matrix B^{-1} (i.e., the solution matrix).

ECCGenerate

The function ECCGenerate is used to generate check codes (that is, the check data matrix J) for a particular configuration of data drives and check drives, using Sequencer 1 and the Parallel Lookup Multiplier as described above. Prior to calling ECCGenerate, ECCSolve is called to compute the appropriate constants for the particular configuration of data drives and check drives, as well as the solution matrix B^{-1} .

ECCRegenerate

The function ECCRegenerate is used to regenerate data vectors and check code vectors for a particular configuration of data drives and check drives (that is, reconstructing the original data matrix D from the surviving data matrix X and the surviving check matrix W, as well as regenerating the lost check data from the restored original data), this time using Sequencer 2 and the Parallel Lookup Multiplier as described above. Prior to calling ECCRegenerate, ECCSolve is called to compute the appropriate constants for the particular configuration of data drives, check drives, and failed drives, as well as the solution matrix B^{-1} .

Exemplary Implementation Details

As discussed in Appendix A, there are two significant sources of computational overhead in erasure code processing (such as an erasure coding system used in RAID processing): the computation of the solution matrix B^{-1} for a given failure scenario, and the byte-level processing of encoding the check data J and reconstructing the lost data after a lost packet (e.g., data drive failure). By reducing the solution matrix B^{-1} to a matrix inversion of a $F \times F$ matrix, where F is the number of lost packets (e.g., failed drives), that portion of the computational overhead is for all intents and purposes negligible compared to the megabytes (MB), gigabytes (GB), and possibly terabytes (TB) of data that needs to be encoded into check data or reconstructed from the surviving original and check data. Accordingly, the remainder of this section will be devoted to the byte-level encoding and regenerating processing.

As already mentioned, certain practical simplifications can be assumed for most implementations. By using a Galois field of 256 entries, byte-level processing can be used for all of the GF arithmetic. Using the master encoding matrix S described in Appendix A, any combination of up to 127 data drives, 1 parity drive, and 128 check drives can be supported with such a Galois field. While, in general, any combination of data drives and check drives that adds up to 256 total drives is possible, not all combinations provide a parity drive when computed directly. Using the master encoding matrix S, on the other hand, allows all such combinations (including a parity drive) to be built (or simply indexed) from the same such matrix. That is, the appropriate sub-matrix (including the parity drive) can be used for configurations of less than the maximum number of drives.

In addition, using the master encoding matrix S permits further data drives and/or check drives can be added without requiring the recomputing of the IDM E (unlike other pro-

US 9,160,374 B2

27

posed solutions, which recompute E for every change of N or M). Rather, additional indexing of rows and/or columns of the master encoding matrix S will suffice. As discussed above, the use of the parity drive can eliminate or significantly reduce the somewhat complex GF multiplication operations associated with the other check drives and replaces them with simple GF addition (bitwise exclusive OR in binary Galois fields) operations. It should be noted that master encoding matrices with the above properties are possible for any power-of-two number of drives $2^P = N_{max} + M_{max}$ where the maximum number of data drives N_{max} is one less than a power of two (e.g., $N_{max} = 127$ or 63) and the maximum number of check drives M_{max} (including the parity drive) is $2^P - N_{max}$.

As discussed earlier, in an exemplary embodiment of the present invention, a modern x86 architecture is used (being readily available and inexpensive). In particular, this architecture supports 16 XMM registers and the SSE instructions. Each XMM register is 128 bits and is available for special purpose processing with the SSE instructions. Each of these XMM registers holds 16 bytes (8-bit), so four such registers can be used to store 64 bytes of data. Thus, by using SSE instructions (some of which work on different operand sizes, for example, treating each of the XMM registers as containing 16 one-byte operands), 64 bytes of data can be operated at a time using four consecutive SSE instructions (e.g., fetching from memory, storing into memory, zeroing, adding, multiplying), the remaining registers being used for intermediate results and temporary storage. With such an architecture, several routines are useful for optimizing the byte-level performance, including the Parallel Lookup Multiplier, Sequencer 1, and Sequencer 2 discussed above.

While the above description contains many specific embodiments of the invention, these should not be construed as limitations on the scope of the invention, but rather as examples of specific embodiments thereof. Accordingly, the scope of the invention should be determined not by the embodiments illustrated, but by the appended claims and their equivalents.

Glossary of Some Variables

A encoding matrix (F×K), sub-matrix of T

B encoding matrix (F×F), sub-matrix of T

B⁻¹ solution matrix (F×F)

C encoded data matrix

$$((N + M) \times L) = \begin{bmatrix} D \\ J \end{bmatrix}$$

C' surviving encoded data matrix

$$(N \times L) = \begin{bmatrix} X \\ W \end{bmatrix}$$

D original data matrix (N×L)

D' permuted original data matrix

$$(N \times L) = \begin{bmatrix} X \\ Y \end{bmatrix}$$

28

E information dispersal matrix

$$(IDM)((N + M) \times N) = \begin{bmatrix} I_N \\ H \end{bmatrix}$$

F number of failed data drives

G number of failed check drives

H check drive encoding matrix (M×N)

I identity matrix ($I_K = K \times K$ identity matrix, $I_N = N \times N$ identity matrix)

J encoded check data matrix (M×L)

K number of surviving data drives = N - F

L data block size (elements or bytes)

M number of check drives

M_{max} maximum value of M

N number of data drives

N_{max} maximum value of N

O zero matrix (K×F), sub-matrix of T

S master encoding matrix $((M_{max} + N_{max}) \times N_{max})$

T transformed IDM

$$(N \times N) = \begin{bmatrix} I_K & O \\ A & B \end{bmatrix}$$

W surviving check data matrix (F×L)

X surviving original data matrix (K×L)

Y lost original data matrix (F×L)

What is claimed is:

1. A system for accelerated error-correcting code (ECC) processing comprising:

a processing core for executing computer instructions and accessing data from a main memory, the processing core comprising at least 16 data registers, each of the data registers comprising at least 16 bytes; and

a non-volatile storage medium for storing the computer instructions,

wherein the processing core, the non-volatile storage medium, and the computer instructions are configured to implement an erasure coding system comprising:

a data matrix for holding original data in the main memory;

a check matrix for holding check data in the main memory;

an encoding matrix for holding first factors in the main memory, the first factors being for encoding the original data into the check data; and

a thread for executing on the processing core and comprising:

a parallel multiplier for concurrently multiplying multiple data entries of a matrix by a single factor; and

a first sequencer for ordering operations through the data matrix and the encoding matrix using the parallel multiplier to generate the check data.

2. The system of claim 1, wherein the parallel multiplier is configured to process the data in units of at least 64 bytes spread over at least four of the data registers at a time.

3. The system of claim 2, wherein the parallel multiplier is further configured to:

receive an input operand in the at least four of the data registers; and

return with the input operand intact in the at least four of the data registers.

4. The system of claim 2, wherein consecutive ones of the computer instructions to process each of the units of the data

US 9,160,374 B2

29

access separate ones of the data registers to permit concurrent execution of the consecutive ones of the computer instructions on the processing core.

5. The system of claim 1, wherein the parallel multiplier comprises two lookup tables for doing concurrent multiplication of 4-bit quantities across 16 byte-sized entries using the PSHUFB (Packed Shuffle Bytes) or equivalent instruction.

6. The system of claim 1, wherein the parallel multiplier is further configured to:

receiving an input operand in at least one of the data registers; and

return with the input operand intact in the at least one of the data registers.

7. A method of accelerated error-correcting code (ECC) processing on a computing system comprising a non-volatile storage medium, a processing core for accessing instructions and data from a main memory, and a computer program comprising a plurality of computer instructions for implementing an erasure coding system, the processing core comprising at least 16 data registers, each of the data registers comprising at least 16 bytes, the method comprising:

storing the computer program on the non-volatile storage medium;

executing the computer instructions on the processing core;

arranging original data as a data matrix in the main memory;

arranging first factors as an encoding matrix in the main memory, the first factors being for encoding the original data into check data, the check data being arranged as a check matrix in the main memory; and

generating the check data using a parallel multiplier for concurrently multiplying multiple data entries of a matrix by a single factor, the generating of the check data comprising ordering operations through the data matrix and the encoding matrix using the parallel multiplier.

8. The method of claim 7, wherein the generating of the check data further comprises processing the data by the parallel multiplier in units of at least 64 bytes spread over at least four of the data registers at a time.

9. The method of claim 8, wherein the generating of the check data further comprises:

receiving by the parallel multiplier an input operand in the at least four of the data registers; and

returning by the parallel multiplier the input operand intact in the at least four of the data registers.

10. The method of claim 8, wherein

consecutive ones of the computer instructions that process each of the units of the data access separate ones of the data registers,

the executing of the computer instructions on the processing core further comprises concurrently executing the consecutive ones of the computer instructions on the processing core.

11. The method of claim 7, wherein the parallel multiplier comprises two lookup tables and the generating of the check data further comprises using the parallel multiplier with the two lookup tables to do concurrent multiplication of 4-bit

30

quantities across 16 byte-sized entries using the PSHUFB (Packed Shuffle Bytes) or equivalent instruction.

12. The method of claim 7, wherein the generating of the check data further comprises:

receiving by the parallel multiplier an input operand in at least one of the data registers; and

returning by the parallel multiplier the input operand intact in the at least one of the data registers.

13. A non-transitory computer-readable storage medium containing a computer program comprising a plurality of computer instructions for performing accelerated error-correcting code (ECC) processing on a computing system comprising a processing core for accessing instructions and data from a main memory, the processing core comprising at least 16 data registers, each of the data registers comprising at least 16 bytes, the computer instructions being configured to implement an erasure coding system when executed on the computing system by performing the steps of:

arranging original data as a data matrix in the main memory;

arranging first factors as an encoding matrix in the main memory, the first factors being for encoding the original data into check data, the check data being arranged as a check matrix in the main memory; and

generating the check data using a parallel multiplier for concurrently multiplying multiple data entries of a matrix by a single factor, the generating of the check data comprising ordering operations through the data matrix and the encoding matrix using the parallel multiplier.

14. The storage medium of claim 13, wherein the generating of the check data further comprises processing the data by the parallel multiplier in units of at least 64 bytes spread over at least four of the data registers at a time.

15. The storage medium of claim 14, wherein the generating of the check data further comprises:

receiving by the parallel multiplier an input operand in the at least four of the data registers; and

returning by the parallel multiplier the input operand intact in the at least four of the data registers.

16. The storage medium of claim 14, wherein

consecutive ones of the computer instructions that process each of the units of the data access separate ones of the data registers,

the executing of the computer instructions on the processing core further comprises concurrently executing the consecutive ones of the computer instructions on the processing core.

17. The storage medium of claim 13, wherein the parallel multiplier comprises two lookup tables and the generating of the check data further comprises using the parallel multiplier with the two lookup tables to do concurrent multiplication of 4-bit quantities across 16 byte-sized entries using the PSHUFB (Packed Shuffle Bytes) or equivalent instruction.

18. The storage medium of claim 13, wherein the generating of the check data further comprises:

receiving by the parallel multiplier an input operand in at least one of the data registers; and

returning by the parallel multiplier the input operand intact in the at least one of the data registers.

* * * * *

EXHIBIT C



US009385759B2

(12) **United States Patent**
Anderson

(10) **Patent No.:** **US 9,385,759 B2**
(45) **Date of Patent:** ***Jul. 5, 2016**

(54) **ACCELERATED ERASURE CODING SYSTEM AND METHOD**

(71) Applicant: **STREAMSCALE, INC.**, Los Angeles, CA (US)

(72) Inventor: **Michael H. Anderson**, Los Angeles, CA (US)

(73) Assignee: **STREAMSCALE, INC.**, Los Angeles, CA (US)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 0 days.

This patent is subject to a terminal disclaimer.

(21) Appl. No.: **14/852,438**

(22) Filed: **Sep. 11, 2015**

(65) **Prior Publication Data**

US 2016/0072525 A1 Mar. 10, 2016

Related U.S. Application Data

(63) Continuation of application No. 14/223,740, filed on Mar. 24, 2014, now Pat. No. 9,160,374, which is a continuation of application No. 13/341,833, filed on Dec. 30, 2011, now Pat. No. 8,683,296.

(51) **Int. Cl.**

H03M 13/00 (2006.01)

H03M 13/37 (2006.01)

(Continued)

(52) **U.S. Cl.**

CPC **H03M 13/616** (2013.01); **G06F 11/1076** (2013.01); **G06F 11/1092** (2013.01);

(Continued)

(58) **Field of Classification Search**

CPC H03M 13/373; H03M 13/3761; H03M 13/3776; H03M 13/616; H03M 13/1191;

H03M 13/134; H03M 13/1515; H04L 1/0043; H04L 1/0057; G06F 11/1076; G06F 11/1092; G06F 11/1096; G06F 12/0238; G06F 12/06; G06F 2211/1057; G06F 2211/109
USPC 714/6.24, 6.1, 6.11, 6.2, 6.21, 6.32, 714/763, 752, 758, 768, 770, 773, 784, 786
See application file for complete search history.

(56) **References Cited**

U.S. PATENT DOCUMENTS

6,654,924 B1 * 11/2003 Hassner G11B 20/183 714/758
6,823,425 B2 * 11/2004 Ghosh G06F 11/1076 711/114

(Continued)

OTHER PUBLICATIONS

Hafner et al., Matrix Methods for Lost Data Reconstruction in Erasure Codes, Nov. 16, 2005, USENIX FAST '05 Paper, pp. 1-26.*

(Continued)

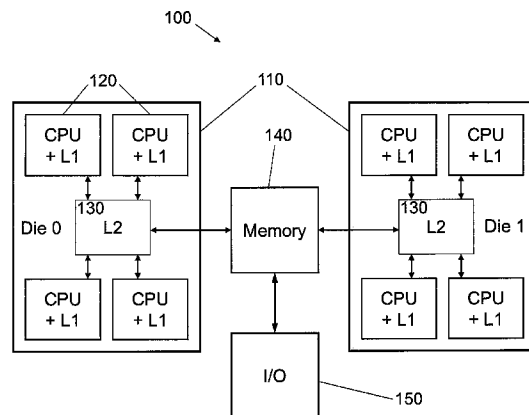
Primary Examiner — John J Tabone, Jr.

(74) *Attorney, Agent, or Firm* — Lewis Roca Rothgerber Christie LLP

(57) **ABSTRACT**

An accelerated erasure coding system includes a processing core for executing computer instructions and accessing data from a main memory, and a non-volatile storage medium for storing the computer instructions. The processing core, storage medium, and computer instructions are configured to implement an erasure coding system, which includes: a data matrix for holding original data in the main memory; a check matrix for holding check data in the main memory; an encoding matrix for holding first factors in the main memory, the first factors being for encoding the original data into the check data; and a thread for executing on the processing core. The thread includes: a parallel multiplier for concurrently multiplying multiple entries of the data matrix by a single entry of the encoding matrix; and a first sequencer for ordering operations through the data matrix and the encoding matrix using the parallel multiplier to generate the check data.

20 Claims, 9 Drawing Sheets



US 9,385,759 B2

Page 2

(51)	<p>Int. Cl. <i>H03M 13/13</i> (2006.01) <i>H04L 1/00</i> (2006.01) <i>G06F 11/10</i> (2006.01) <i>G06F 12/02</i> (2006.01) <i>G06F 12/06</i> (2006.01) <i>H03M 13/11</i> (2006.01) <i>H03M 13/15</i> (2006.01)</p>	<p>9,160,374 B2 * 10/2015 Anderson H03M 13/373 714/763 2011/0029756 A1 * 2/2011 Biscondi H03M 13/1114 712/22 2012/0272036 A1 * 10/2012 Muralimanohar .. G06F 12/0238 711/202 2013/0108048 A1 * 5/2013 Grube H04W 12/00 380/270 2013/0110962 A1 * 5/2013 Grube H04W 12/00 709/213 2013/0111552 A1 * 5/2013 Grube H04Q 12/00 726/3 2013/0124932 A1 * 5/2013 Schuh G06F 9/44 714/718 2013/0173956 A1 * 7/2013 Anderson G06F 11/1076 714/6.24 2013/0173996 A1 * 7/2013 Anderson H03M 13/134 714/770 2015/0012796 A1 * 1/2015 Anderson H03M 13/134 714/763</p>
(52)	<p>U.S. Cl. CPC <i>G06F11/1096</i> (2013.01); <i>G06F 12/0238</i> (2013.01); <i>G06F 12/06</i> (2013.01); <i>H03M</i> <i>13/1191</i> (2013.01); <i>H03M 13/134</i> (2013.01); <i>H03M 13/1515</i> (2013.01); <i>H03M 13/373</i> (2013.01); <i>H03M 13/3761</i> (2013.01); <i>H03M</i> <i>13/3776</i> (2013.01); <i>H04L 1/0043</i> (2013.01); <i>H04L 1/0057</i> (2013.01); <i>G06F 2211/1057</i> (2013.01)</p>	

(56) **References Cited**

U.S. PATENT DOCUMENTS

7,350,126 B2 *	3/2008	Winograd	G06F 11/1076 714/752
7,930,337 B2	4/2011	Hasenplaugh et al.	
8,145,941 B2 *	3/2012	Jacobson	G06F 11/1076 714/6.24
8,352,847 B2 *	1/2013	Gunnam	G06F 17/16 714/758
8,683,296 B2 *	3/2014	Anderson	H03M 13/134 714/6.24

OTHER PUBLICATIONS

Anvin; The mathematics of Raid-6; First Version Jan. 20, 2004; Last Updated Dec. 20, 2011; pp. 1-9.
Maddock, et al.; White Paper, Surviving Two Disk Failures Introducing Various "RAID 6" Implementations; Xyratex; pp. 1-13.
Plank; All About Erasure Codes:—Reed-Solomon Coding—LDPC Coding; Logistical Computing and Internetworking Laboratory, Department of Computer Science, University of Tennessee; ICL—Aug. 20, 2004; 52 sheets.

* cited by examiner

FIG. 1

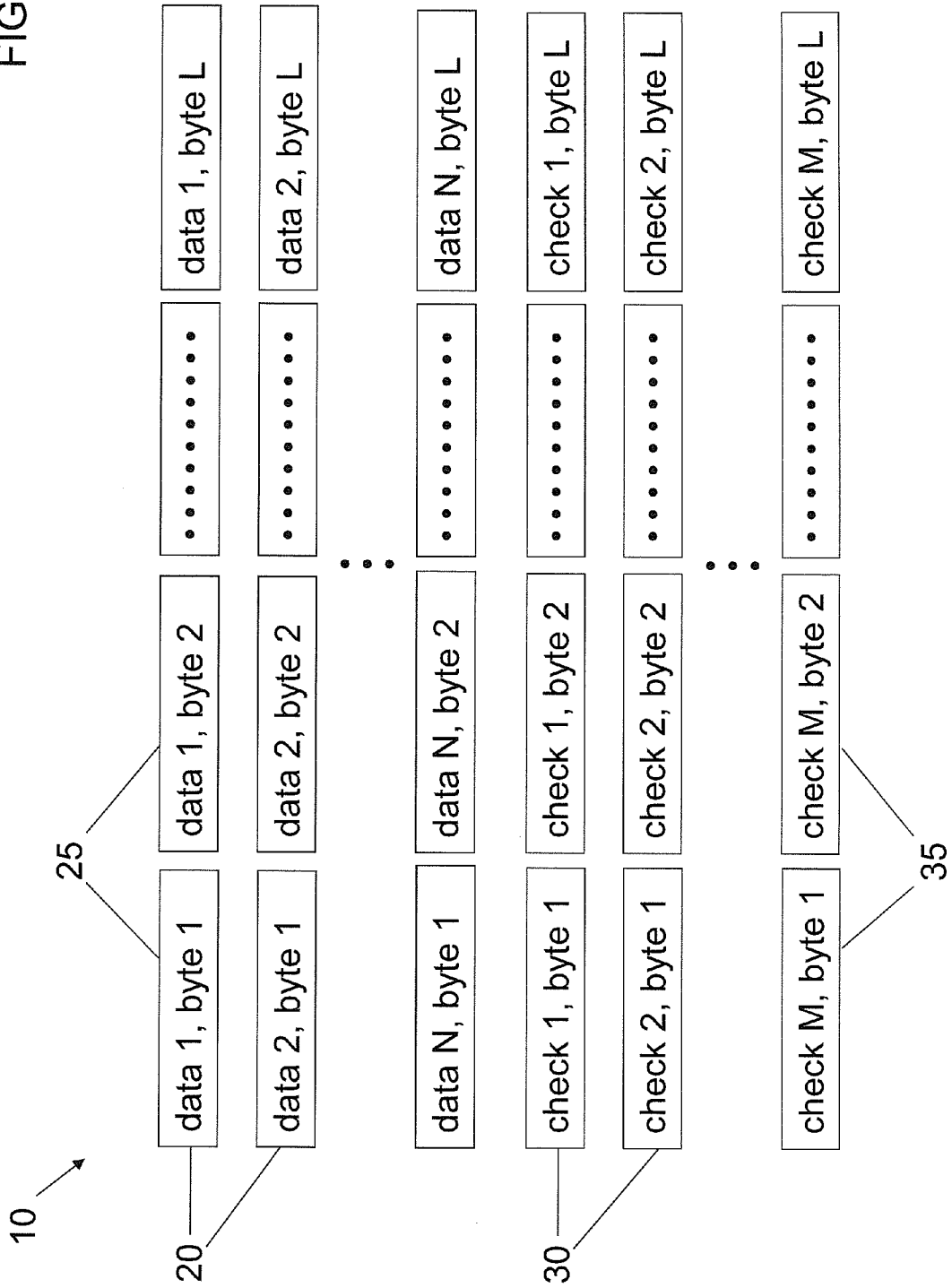


FIG. 2

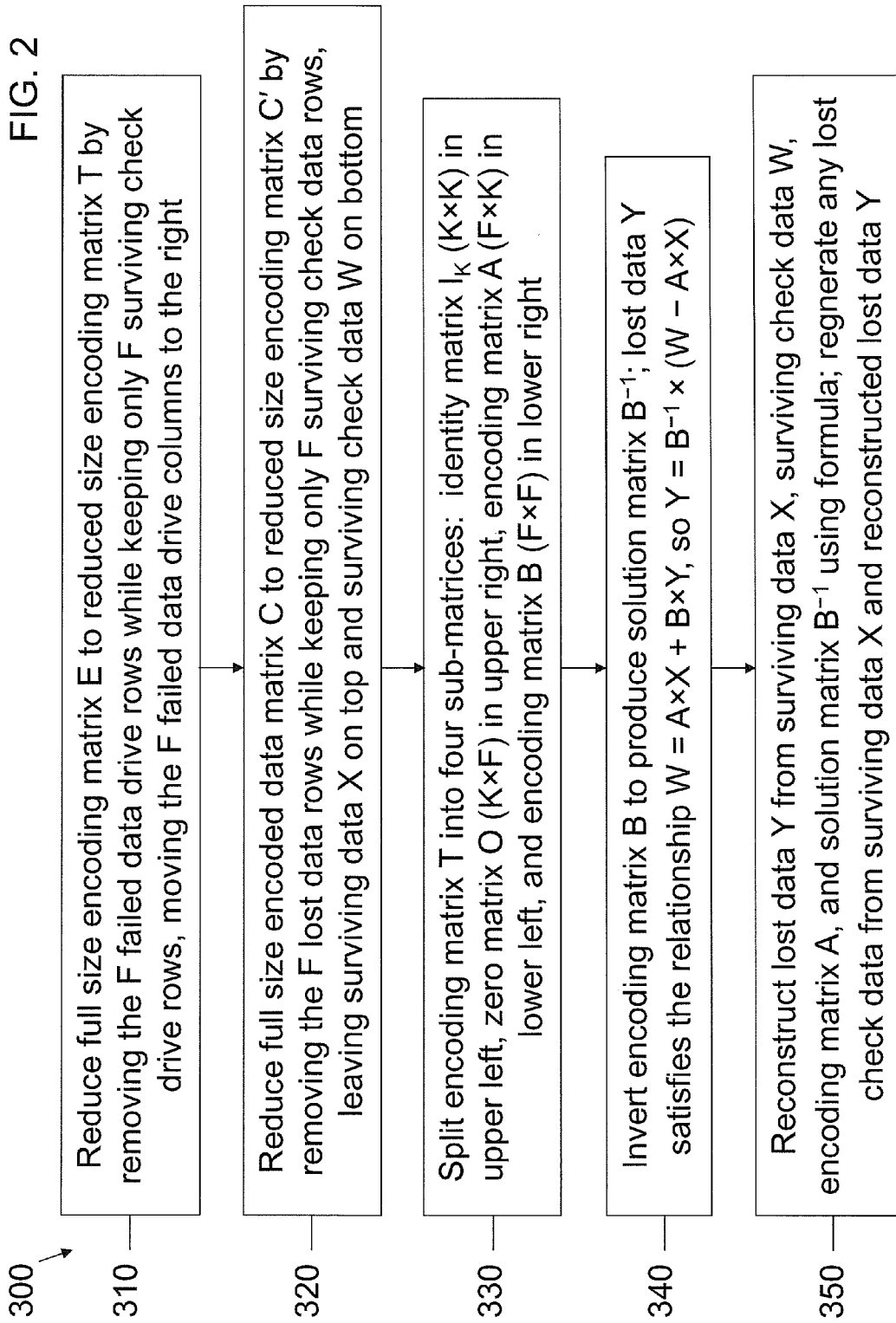


FIG. 3

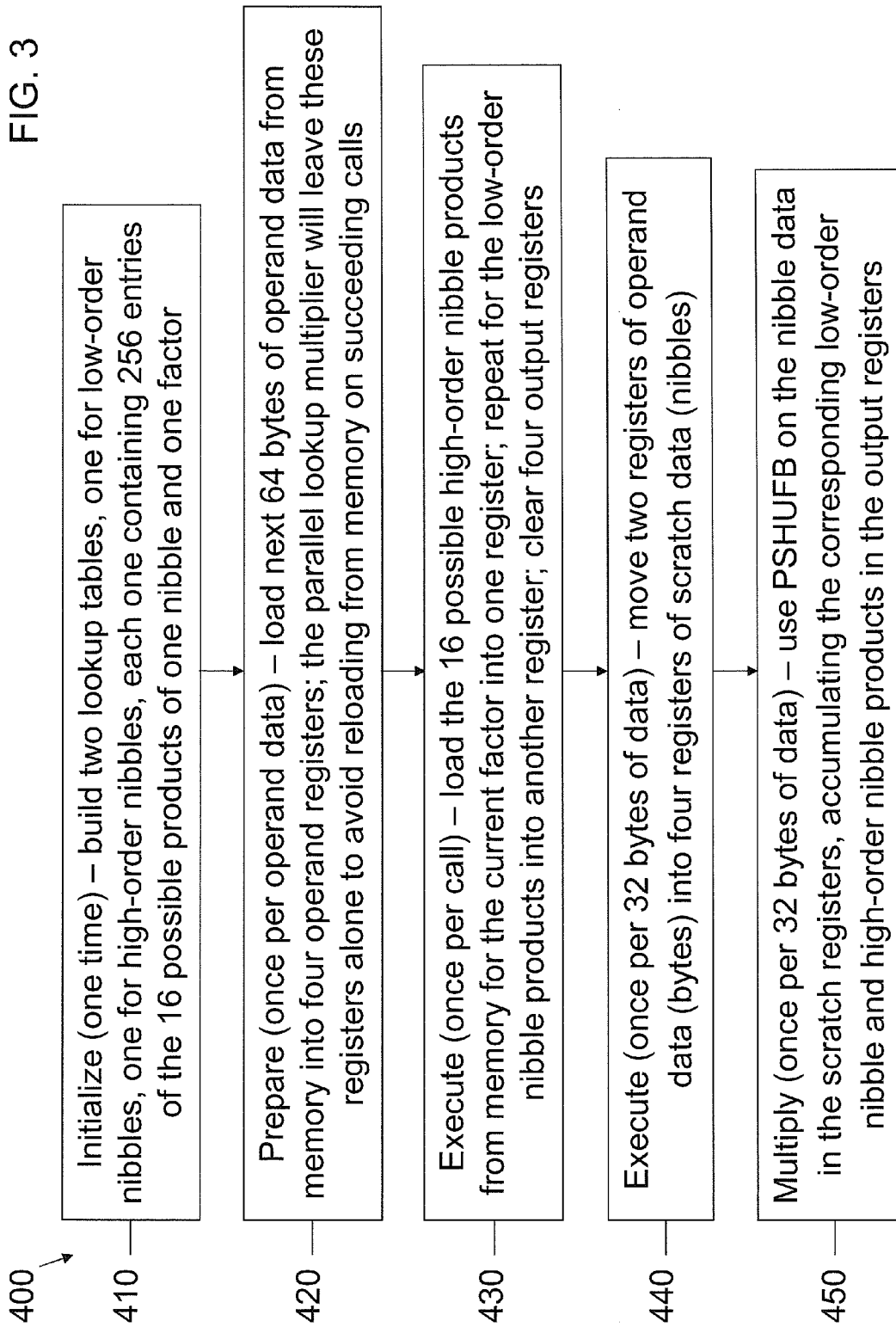


FIG. 4

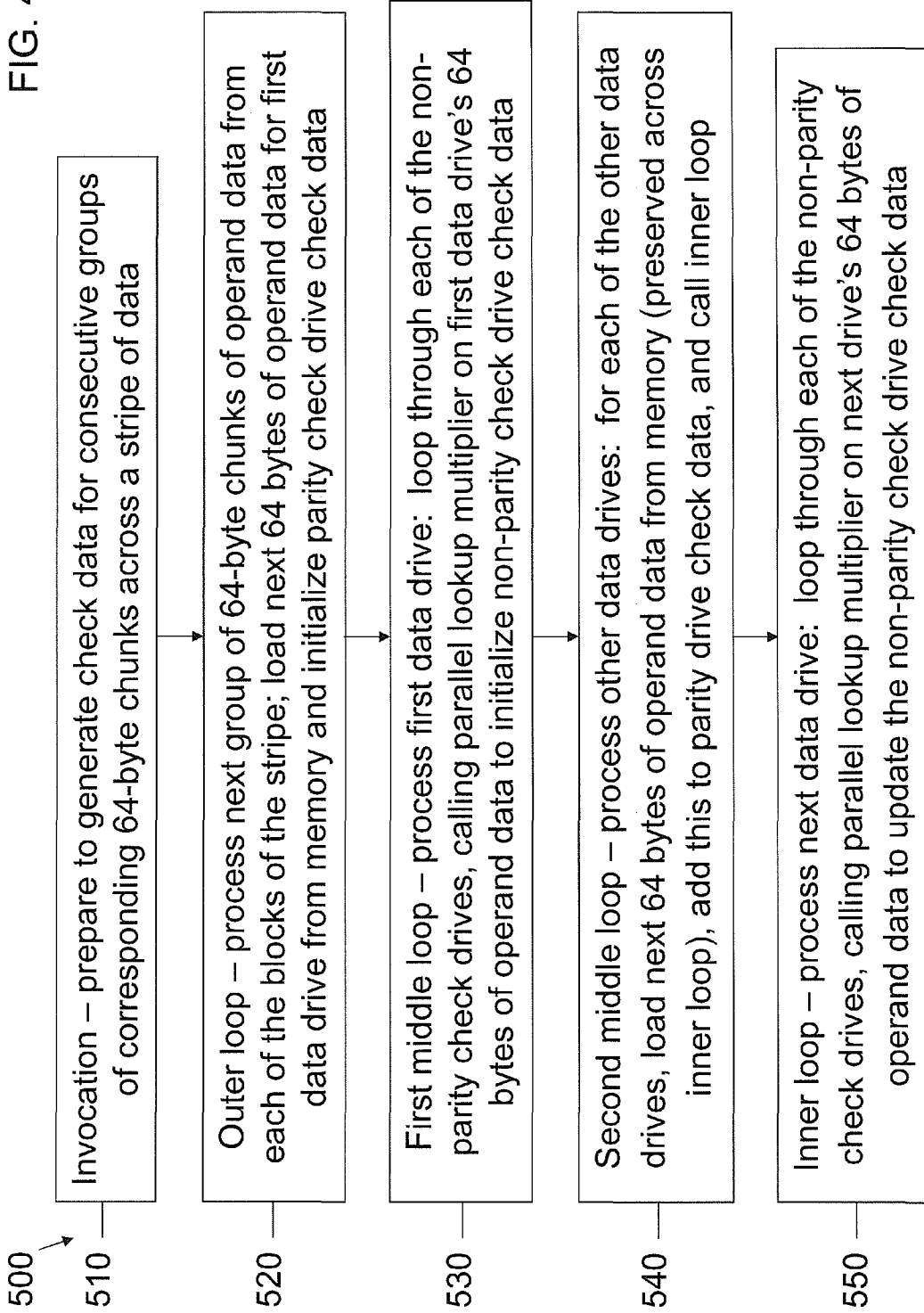


FIG. 5

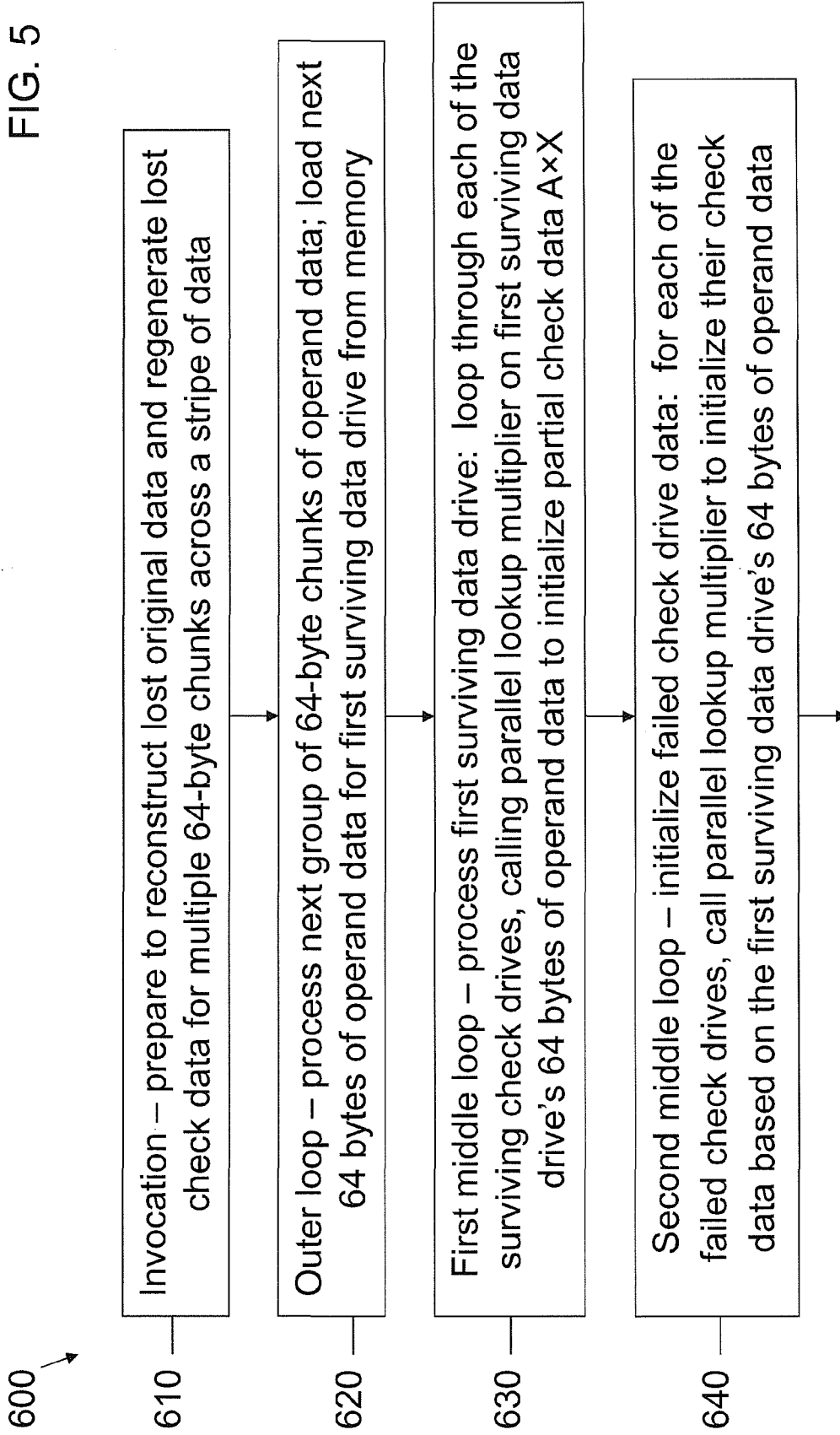


FIG. 6

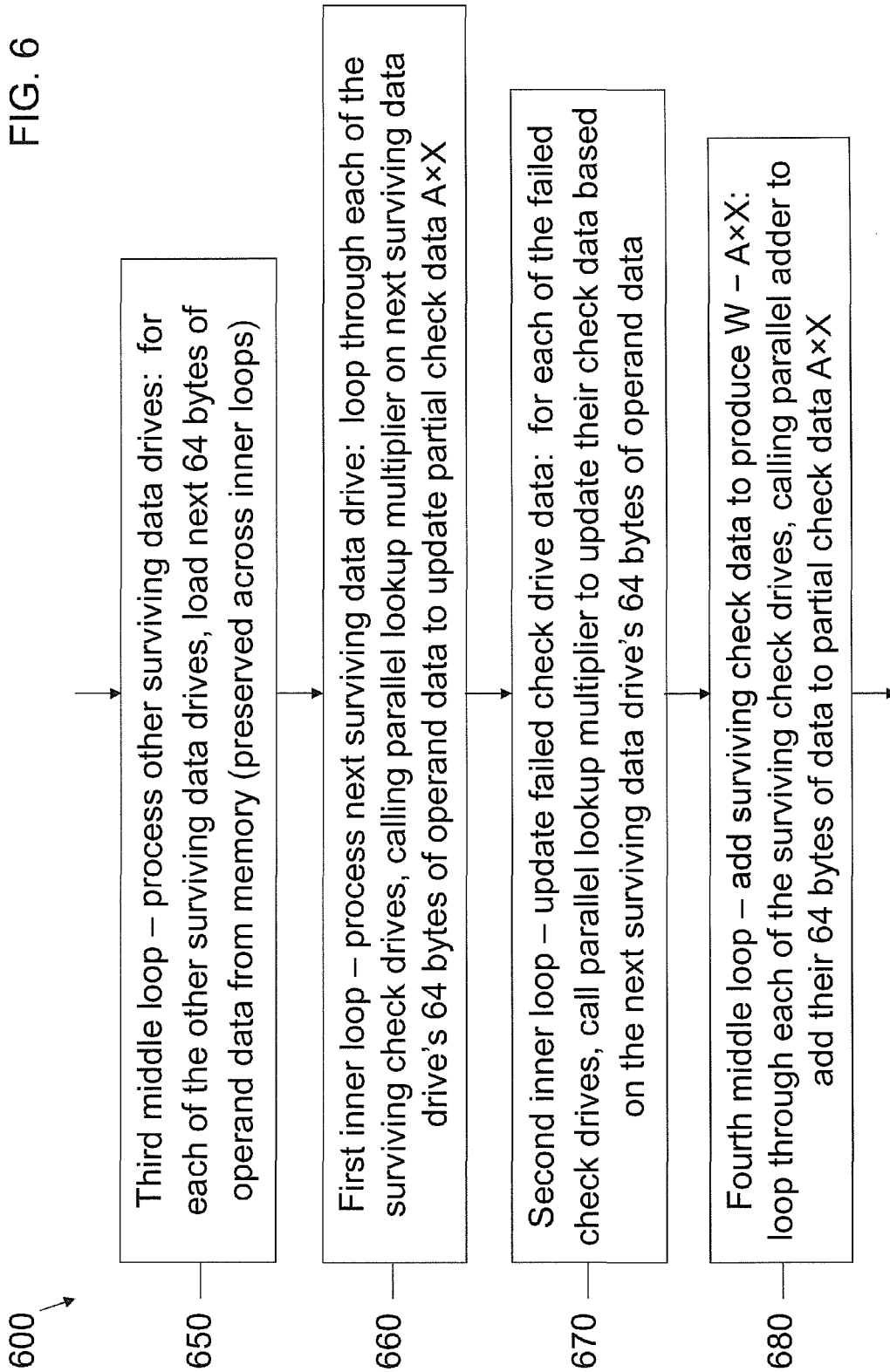


FIG. 7

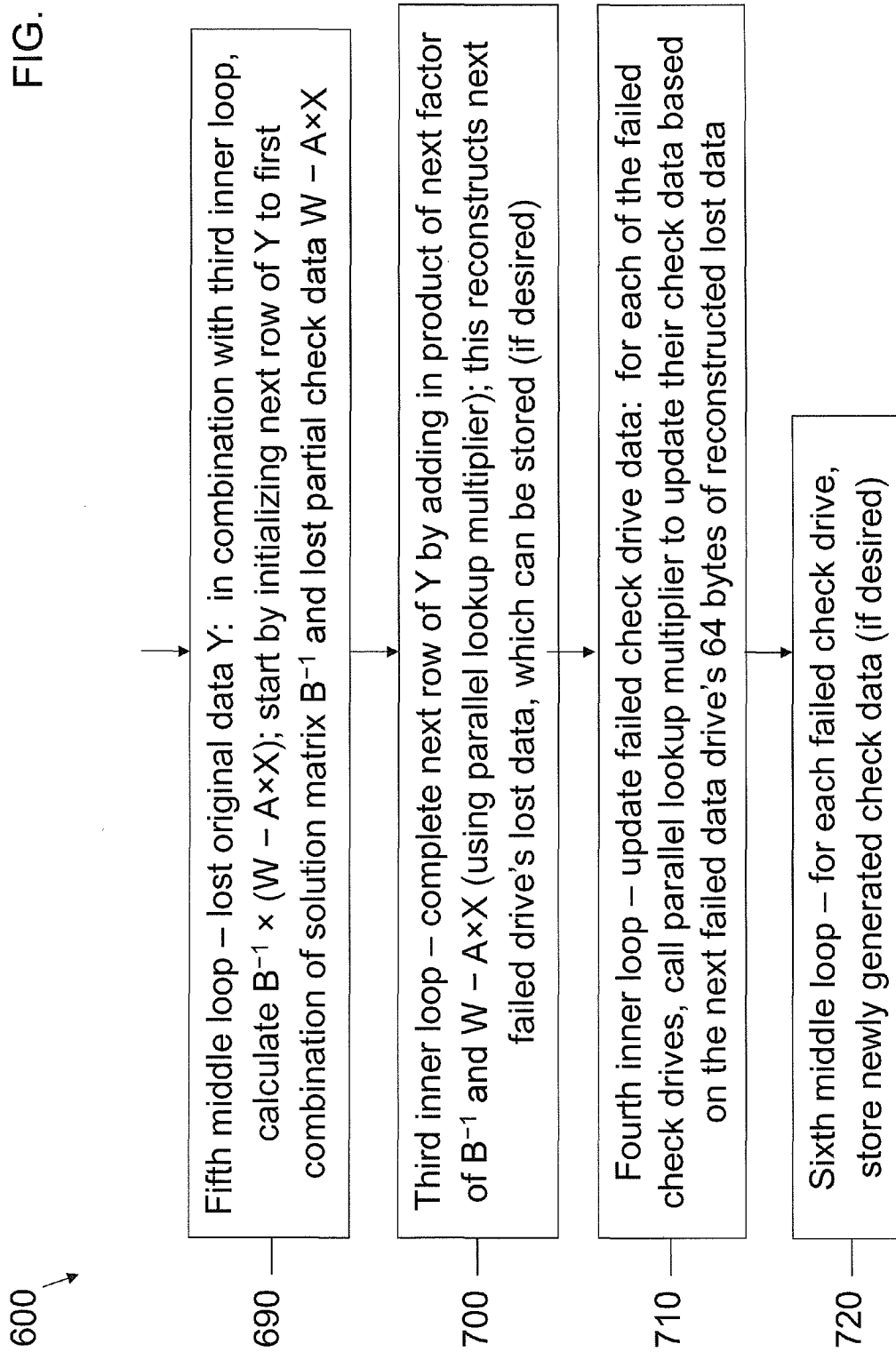


FIG. 8

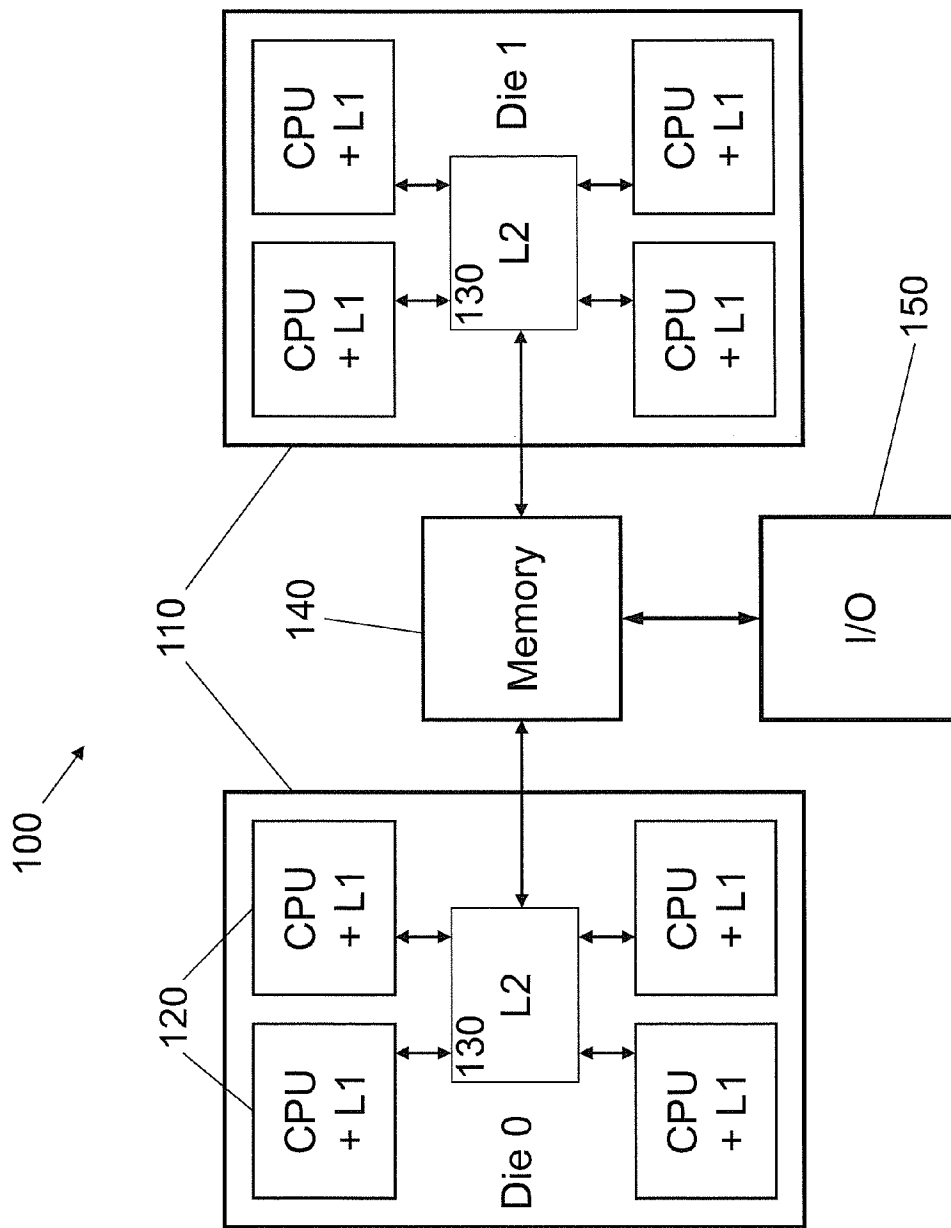
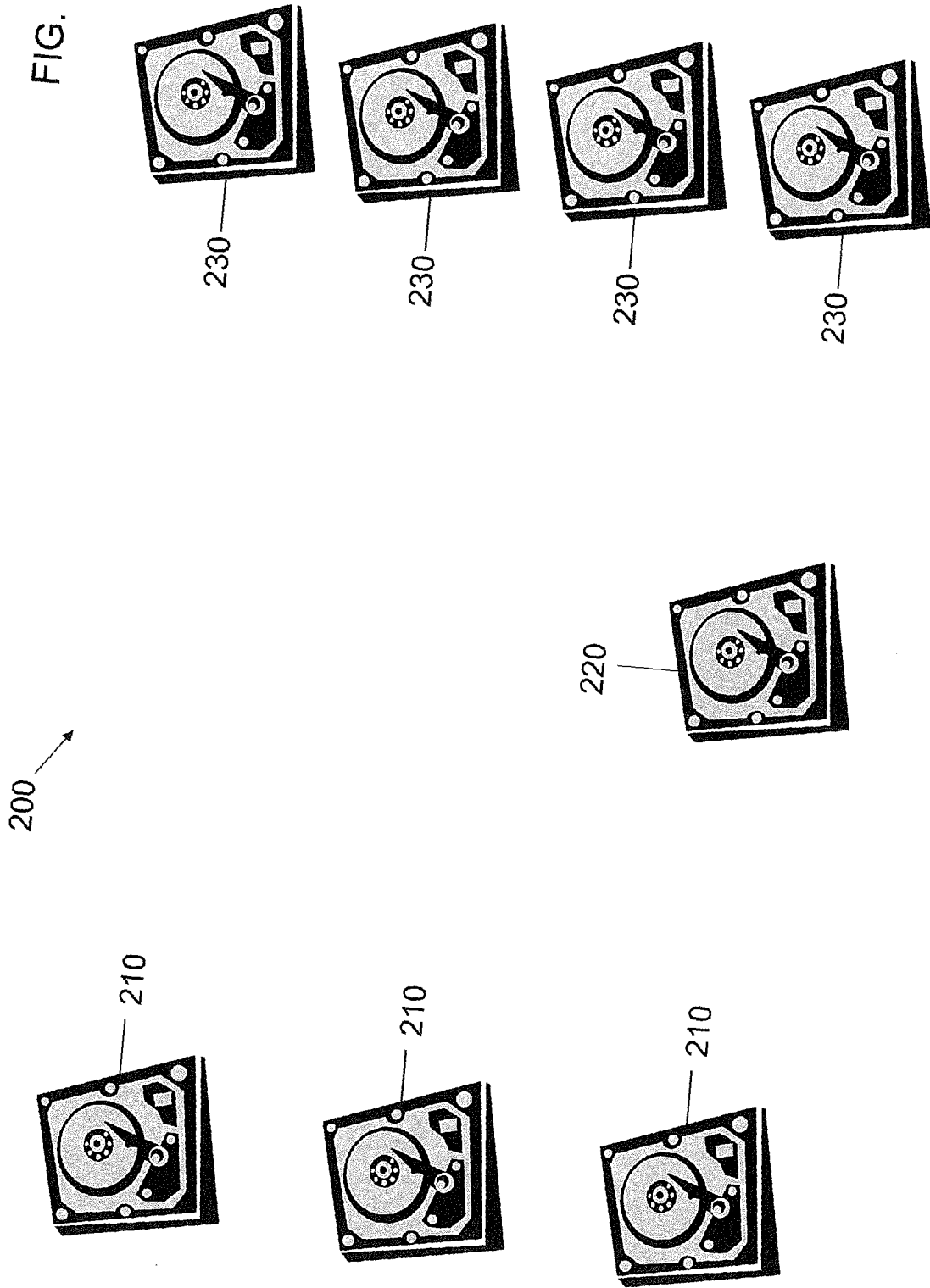


FIG. 9



US 9,385,759 B2

1

ACCELERATED ERASURE CODING SYSTEM AND METHOD

CROSS-REFERENCE TO RELATED APPLICATIONS

This application is a continuation of U.S. patent application Ser. No. 14/223,740, filed on Mar. 24, 2014, which is a continuation of U.S. patent application Ser. No. 13/341,833, filed on Dec. 30, 2011, now U.S. Pat. No. 8,683,296, issued on Mar. 25, 2014, the entire contents of each of which are expressly incorporated herein by reference.

BACKGROUND

1. Field

Aspects of embodiments of the present invention are directed toward an accelerated erasure coding system and method.

2. Description of Related Art

An erasure code is a type of error-correcting code (ECC) useful for forward error-correction in applications like a redundant array of independent disks (RAID) or high-speed communication systems. In a typical erasure code, data (or original data) is organized in stripes, each of which is broken up into N equal-sized blocks, or data blocks, for some positive integer N . The data for each stripe is thus reconstructable by putting the N data blocks together. However, to handle situations where one or more of the original N data blocks gets lost, erasure codes also encode an additional M equal-sized blocks (called check blocks or check data) from the original N data blocks, for some positive integer M .

The N data blocks and the M check blocks are all the same size. Accordingly, there are a total of $N+M$ equal-sized blocks after encoding. The $N+M$ blocks may, for example, be transmitted to a receiver as $N+M$ separate packets, or written to $N+M$ corresponding disk drives. For ease of description, all $N+M$ blocks after encoding will be referred to as encoded blocks, though some (for example, N of them) may contain unencoded portions of the original data. That is, the encoded data refers to the original data together with the check data.

The M check blocks build redundancy into the system, in a very efficient manner, in that the original data (as well as any lost check data) can be reconstructed if any N of the $N+M$ encoded blocks are received by the receiver, or if any N of the $N+M$ disk drives are functioning correctly. Note that such an erasure code is also referred to as "optimal." For ease of description, only optimal erasure codes will be discussed in this application. In such a code, up to M of the encoded blocks can be lost, (e.g., up to M of the disk drives can fail) so that if any N of the $N+M$ encoded blocks are received successfully by the receiver, the original data (as well as the check data) can be reconstructed. $N/(N+M)$ is thus the code rate of the erasure code encoding (i.e., how much space the original data takes up in the encoded data). Erasure codes for select values of N and M can be implemented on RAID systems employing $N+M$ (disk) drives by spreading the original data among N "data" drives, and using the remaining M drives as "check" drives. Then, when any N of the $N+M$ drives are correctly functioning, the original data can be reconstructed, and the check data can be regenerated.

Erasure codes (or more specifically, erasure coding systems) are generally regarded as impractical for values of M larger than 1 (e.g., RAID5 systems, such as parity drive systems) or 2 (RAID6 systems), that is, for more than one or two

2

rated herein by reference, p. 7, "Thus, in 2-disk-degraded mode, performance will be very slow. However, it is expected that that will be a rare occurrence, and that performance will not matter significantly in that case." See also Robert Maddock et al., "Surviving Two Disk Failures," p. 6, "The main difficulty with this technique is that calculating the check codes, and reconstructing data after failures, is quite complex. It involves polynomials and thus multiplication, and requires special hardware, or at least a signal processor, to do it at sufficient speed." In addition, see also James S. Plank, "All About Erasure Codes: —Reed-Solomon Coding—LDPC Coding," slide 15 (describing computational complexity of Reed-Solomon decoding), "Bottom line: When n & m grow, it is brutally expensive." Accordingly, there appears to be a general consensus among experts in the field that erasure coding systems are impractical for RAID systems for all but small values of M (that is, small numbers of check drives), such as 1 or 2.

Modern disk drives, on the other hand, are much less reliable than those envisioned when RAID was proposed. This is due to their capacity growing out of proportion to their reliability. Accordingly, systems with only a single check disk have, for the most part, been discontinued in favor of systems with two check disks.

In terms of reliability, a higher check disk count is clearly more desirable than a lower check disk count. If the count of error events on different drives is larger than the check disk count, data may be lost and that cannot be reconstructed from the correctly functioning drives. Error events extend well beyond the traditional measure of advertised mean time between failures (MTBF). A simple, real world example is a service event on a RAID system where the operator mistakenly replaces the wrong drive or, worse yet, replaces a good drive with a broken drive. In the absence of any generally accepted methodology to train, certify, and measure the effectiveness of service technicians, these types of events occur at an unknown rate, but certainly occur. The foolproof solution for protecting data in the face of multiple error events is to increase the check disk count.

SUMMARY

Aspects of embodiments of the present invention address these problems by providing a practical erasure coding system that, for byte-level RAID processing (where each byte is made up of 8 bits), performs well even for values of $N+M$ as large as 256 drives (for example, $N=127$ data drives and $M=129$ check drives). Further aspects provide for a single precomputed encoding matrix (or master encoding matrix) S of size $M_{max} \times N_{max}$ or $(N_{max} + M_{max}) \times N_{max}$ or $(M_{max} - 1) \times N_{max}$ elements (e.g., bytes), which can be used, for example, for any combination of $N \leq N_{max}$ data drives and $M \leq M_{max}$ check drives such that $N_{max} + M_{max} \leq 256$ (e.g., $N_{max} = 127$ and $M_{max} = 129$, or $N_{max} = 63$ and $M_{max} = 193$). This is an improvement over prior art solutions that rebuild such matrices from scratch every time N or M changes (such as adding another check drive). Still higher values of N and M are possible with larger processing increments, such as 2 bytes, which affords up to $N+M=65,536$ drives (such as $N=32,767$ data drives and $M=32,769$ check drives).

Higher check disk count can offer increased reliability and decreased cost. The higher reliability comes from factors such as the ability to withstand more drive failures. The decreased cost arises from factors such as the ability to create larger groups of data drives. For example, systems with two checks disks are typically limited to group sizes of 10 or fewer drives for reliability reasons. With a higher check disk count,

US 9,385,759 B2

3

larger groups are available, which can lead to fewer overall components for the same unit of storage and hence, lower cost.

Additional aspects of embodiments of the present invention further address these problems by providing a standard parity drive as part of the encoding matrix. For instance, aspects provide for a parity drive for configurations with up to 127 data drives and up to 128 (non-parity) check drives, for a total of up to 256 total drives including the parity drive. Further aspects provide for different breakdowns, such as up to 63 data drives, a parity drive, and up to 192 (non-parity) check drives. Providing a parity drive offers performance comparable to RAID5 in comparable circumstances (such as single data drive failures) while also being able to tolerate significantly larger numbers of data drive failures by including additional (non-parity) check drives.

Further aspects are directed to a system and method for implementing a fast solution matrix algorithm for Reed-Solomon codes. While known solution matrix algorithms compute an $N \times N$ solution matrix (see, for example, J. S. Plank, "A tutorial on Reed-Solomon coding for fault-tolerance in RAID-like systems," *Software—Practice & Experience*, 27(9):995-1012, September 1997, and J. S. Plank and Y. Ding, "Note: Correction to the 1997 tutorial on Reed-Solomon coding," Technical Report CS-03-504, University of Tennessee, April 2003), requiring $O(N^3)$ operations, regardless of the number of failed data drives, aspects of embodiments of the present invention compute only an $F \times F$ solution matrix, where F is the number of failed data drives. The overhead for computing this $F \times F$ solution matrix is approximately $F^3/3$ multiplication operations and the same number of addition operations. Not only is $F \leq N$, in almost any practical application, the number of failed data drives F is considerably smaller than the number of data drives N . Accordingly, the fast solution matrix algorithm is considerably faster than any known approach for practical values of F and N .

Still further aspects are directed toward fast implementations of the check data generation and the lost (original and check) data reconstruction. Some of these aspects are directed toward fetching the surviving (original and check) data a minimum number of times (that is, at most once) to carry out the data reconstruction. Some of these aspects are directed toward efficient implementations that can maximize or significantly leverage the available parallel processing power of multiple cores working concurrently on the check data generation and the lost data reconstruction. Existing implementations do not attempt to accelerate these aspects of the data generation and thus fail to achieve a comparable level of performance.

In an exemplary embodiment of the present invention, a system for accelerated error-correcting code (ECC) processing is provided. The system includes a processing core for executing computer instructions and accessing data from a main memory; and a non-volatile storage medium (for example, a disk drive, or flash memory) for storing the computer instructions. The processing core, the storage medium, and the computer instructions are configured to implement an erasure coding system. The erasure coding system includes a data matrix for holding original data in the main memory, a check matrix for holding check data in the main memory, an encoding matrix for holding first factors in the main memory, and a thread for executing on the processing core. The first factors are for encoding the original data into the check data. The thread includes a parallel multiplier for concurrently multiplying multiple data entries of a matrix by a single factor; and a first sequencer for ordering operations through

4

the data matrix and the encoding matrix using the parallel multiplier to generate the check data.

The first sequencer may be configured to access each entry of the data matrix from the main memory at most once while generating the check data.

The processing core may include a plurality of processing cores. The thread may include a plurality of threads. The erasure coding system may further include a scheduler for generating the check data by dividing the data matrix into a plurality of data matrices, dividing the check matrix into a plurality of check matrices, assigning corresponding ones of the data matrices and the check matrices to the threads, and assigning the threads to the processing cores to concurrently generate portions of the check data corresponding to the check matrices from respective ones of the data matrices.

The data matrix may include a first number of rows. The check matrix may include a second number of rows. The encoding matrix may include the second number of rows and the first number of columns.

The data matrix may be configured to add rows to the first number of rows or the check matrix may be configured to add rows to the second number of rows while the first factors remain unchanged.

Each of entries of one of the rows of the encoding matrix may include a multiplicative identity factor (such as 1).

The data matrix may be configured to be divided by rows into a surviving data matrix for holding surviving original data of the original data, and a lost data matrix corresponding to lost original data of the original data and including a third number of rows. The erasure coding system may further include a solution matrix for holding second factors in the main memory. The second factors are for decoding the check data into the lost original data using the surviving original data and the first factors.

The solution matrix may include the third number of rows and the third number of columns.

The solution matrix may further include an inverted said third number by said third number sub-matrix of the encoding matrix.

The erasure coding system may further include a first list of rows of the data matrix corresponding to the surviving data matrix, and a second list of rows of the data matrix corresponding to the lost data matrix.

The data matrix may be configured to be divided into a surviving data matrix for holding surviving original data of the original data, and a lost data matrix corresponding to lost original data of the original data. The erasure coding system may further include a solution matrix for holding second factors in the main memory. The second factors are for decoding the check data into the lost original data using the surviving original data and the first factors. The thread may further include a second sequencer for ordering operations through the surviving data matrix, the encoding matrix, the check matrix, and the solution matrix using the parallel multiplier to reconstruct the lost original data.

The second sequencer may be further configured to access each entry of the surviving data matrix from the main memory at most once while reconstructing the lost original data.

The processing core may include a plurality of processing cores. The thread may include a plurality of threads. The erasure coding system may further include: a scheduler for generating the check data and reconstructing the lost original data by dividing the data matrix into a plurality of data matrices; dividing the surviving data matrix into a plurality of surviving data matrices; dividing the lost data matrix into a plurality of lost data matrices; dividing the check matrix into a plurality of check matrices; assigning corresponding ones

US 9,385,759 B2

5

of the data matrices, the surviving data matrices, the lost data matrices, and the check matrices to the threads; and assigning the threads to the processing cores to concurrently generate portions of the check data corresponding to the check matrices from respective ones of the data matrices and to concurrently reconstruct portions of the lost original data corresponding to the lost data matrices from respective ones of the surviving data matrices and the check matrices.

The check matrix may be configured to be divided into a surviving check matrix for holding surviving check data of the check data, and a lost check matrix corresponding to lost check data of the check data. The second sequencer may be configured to order operations through the surviving data matrix, the reconstructed lost original data, and the encoding matrix using the parallel multiplier to regenerate the lost check data.

The second sequencer may be further configured to reconstruct the lost original data concurrently with regenerating the lost check data.

The second sequencer may be further configured to access each entry of the surviving data matrix from the main memory at most once while reconstructing the lost original data and regenerating the lost check data.

The second sequencer may be further configured to regenerate the lost check data without accessing the reconstructed lost original data from the main memory.

The processing core may include a plurality of processing cores. The thread may include a plurality of threads. The erasure coding system may further include a scheduler for generating the check data, reconstructing the lost original data, and regenerating the lost check data by: dividing the data matrix into a plurality of data matrices; dividing the surviving data matrix into a plurality of surviving data matrices; dividing the lost data matrix into a plurality of lost data matrices; dividing the check matrix into a plurality of check matrices; dividing the surviving check matrix into a plurality of surviving check matrices; dividing the lost check matrix into a plurality of lost check matrices; assigning corresponding ones of the data matrices, the surviving data matrices, the lost data matrices, the check matrices, the surviving check matrices, and the lost check matrices to the threads; and assigning the threads to the processing cores to concurrently generate portions of the check data corresponding to the check matrices from respective ones of the data matrices, to concurrently reconstruct portions of the lost original data corresponding to the lost data matrices from respective ones of the surviving data matrices and the surviving check matrices, and to concurrently regenerate portions of the lost check data corresponding to the lost check matrices from respective ones of the surviving data matrices and respective portions of the reconstructed lost original data.

The processing core may include 16 data registers. Each of the data registers may include 16 bytes. The parallel multiplier may be configured to process the data in units of at least 64 bytes spread over at least four of the data registers at a time.

Consecutive instructions to process each of the units of the data may access separate ones of the data registers to permit concurrent execution of the consecutive instructions by the processing core.

The parallel multiplier may include two lookup tables for doing concurrent multiplication of 4-bit quantities across 16 byte-sized entries using the PSHUFB (Packed Shuffle Bytes) instruction.

The parallel multiplier may be further configured to receive an input operand in four of the data registers, and return with the input operand intact in the four of the data registers.

6

According to another exemplary embodiment of the present invention, a method of accelerated error-correcting code (ECC) processing on a computing system is provided. The computing system includes a non-volatile storage medium (such as a disk drive or flash memory), a processing core for accessing instructions and data from a main memory, and a computer program including a plurality of computer instructions for implementing an erasure coding system. The method includes: storing the computer program on the storage medium; executing the computer instructions on the processing core; arranging original data as a data matrix in the main memory; arranging first factors as an encoding matrix in the main memory, the first factors being for encoding the original data into check data, the check data being arranged as a check matrix in the main memory; and generating the check data using a parallel multiplier for concurrently multiplying multiple data entries of a matrix by a single factor. The generating of the check data includes ordering operations through the data matrix and the encoding matrix using the parallel multiplier.

The generating of the check data may include accessing each entry of the data matrix from the main memory at most once.

The processing core may include a plurality of processing cores. The executing of the computer instructions may include executing the computer instructions on the processing cores. The method may further include scheduling the generating of the check data by: dividing the data matrix into a plurality of data matrices; dividing the check matrix into a plurality of check matrices; and assigning corresponding ones of the data matrices and the check matrices to the processing cores to concurrently generate portions of the check data corresponding to the check matrices from respective ones of the data matrices.

The method may further include: dividing the data matrix into a surviving data matrix for holding surviving original data of the original data, and a lost data matrix corresponding to lost original data of the original data; arranging second factors as a solution matrix in the main memory, the second factors being for decoding the check data into the lost original data using the surviving original data and the first factors; and reconstructing the lost original data by ordering operations through the surviving data matrix, the encoding matrix, the check matrix, and the solution matrix using the parallel multiplier.

The reconstructing of the lost original data may include accessing each entry of the surviving data matrix from the main memory at most once.

The processing core may include a plurality of processing cores. The executing of the computer instructions may include executing the computer instructions on the processing cores. The method may further include scheduling the generating of the check data and the reconstructing of the lost original data by: dividing the data matrix into a plurality of data matrices; dividing the surviving data matrix into a plurality of surviving data matrices; dividing the lost data matrix into a plurality of lost data matrices; dividing the check matrix into a plurality of check matrices; and assigning corresponding ones of the data matrices, the surviving data matrices, the lost data matrices, and the check matrices to the processing cores to concurrently generate portions of the check data corresponding to the check matrices from respective ones of the data matrices and to concurrently reconstruct portions of the lost original data corresponding to the lost data matrices from respective ones of the surviving data matrices and the check matrices.

US 9,385,759 B2

7

The method may further include: dividing the check matrix into a surviving check matrix for holding surviving check data of the check data, and a lost check matrix corresponding to lost check data of the check data; and regenerating the lost check data by ordering operations through the surviving data matrix, the reconstructed lost original data, and the encoding matrix using the parallel multiplier.

The reconstructing of the lost original data may take place concurrently with the regenerating of the lost check data.

The reconstructing of the lost original data and the regenerating of the lost check data may include accessing each entry of the surviving data matrix from the main memory at most once.

The regenerating of the lost check data may take place without accessing the reconstructed lost original data from the main memory.

The processing core may include a plurality of processing cores. The executing of the computer instructions may include executing the computer instructions on the processing cores. The method may further include scheduling the generating of the check data, the reconstructing of the lost original data, and the regenerating of the lost check data by: dividing the data matrix into a plurality of data matrices; dividing the surviving data matrix into a plurality of surviving data matrices; dividing the lost data matrix into a plurality of lost data matrices; dividing the check matrix into a plurality of check matrices; dividing the surviving check matrix into a plurality of surviving check matrices; dividing the lost check matrix into a plurality of lost check matrices; and assigning corresponding ones of the data matrices, the surviving data matrices, the lost data matrices, the check matrices, the surviving check matrices, and the lost check matrices to the processing cores to concurrently generate portions of the check data corresponding to the check matrices from respective ones of the data matrices, to concurrently reconstruct portions of the lost original data corresponding to the lost data matrices from respective ones of the surviving data matrices and the surviving check matrices, and to concurrently regenerate portions of the lost check data corresponding to the lost check matrices from respective ones of the surviving data matrices and respective portions of the reconstructed lost original data.

According to yet another exemplary embodiment of the present invention, a non-transitory computer-readable storage medium (such as a disk drive, a compact disk (CD), a digital video disk (DVD), flash memory, a universal serial bus (USB) drive, etc.) containing a computer program including a plurality of computer instructions for performing accelerated error-correcting code (ECC) processing on a computing system is provided. The computing system includes a processing core for accessing instructions and data from a main memory. The computer instructions are configured to implement an erasure coding system when executed on the computing system by performing the steps of: arranging original data as a data matrix in the main memory; arranging first factors as an encoding matrix in the main memory, the first factors being for encoding the original data into check data, the check data being arranged as a check matrix in the main memory; and generating the check data using a parallel multiplier for concurrently multiplying multiple data entries of a matrix by a single factor. The generating of the check data includes ordering operations through the data matrix and the encoding matrix using the parallel multiplier.

The generating of the check data may include accessing each entry of the data matrix from the main memory at most once.

8

The processing core may include a plurality of processing cores. The computer instructions may be further configured to perform the step of scheduling the generating of the check data by: dividing the data matrix into a plurality of data matrices; dividing the check matrix into a plurality of check matrices; and assigning corresponding ones of the data matrices and the check matrices to the processing cores to concurrently generate portions of the check data corresponding to the check matrices from respective ones of the data matrices.

The computer instructions may be further configured to perform the steps of: dividing the data matrix into a surviving data matrix for holding surviving original data of the original data, and a lost data matrix corresponding to lost original data of the original data; arranging second factors as a solution matrix in the main memory, the second factors being for decoding the check data into the lost original data using the surviving original data and the first factors; and reconstructing the lost original data by ordering operations through the surviving data matrix, the encoding matrix, the check matrix, and the solution matrix using the parallel multiplier.

The computer instructions may be further configured to perform the steps of: dividing the check matrix into a surviving check matrix for holding surviving check data of the check data, and a lost check matrix corresponding to lost check data of the check data; and regenerating the lost check data by ordering operations through the surviving data matrix, the reconstructed lost original data, and the encoding matrix using the parallel multiplier.

The reconstructing of the lost original data and the regenerating of the lost check data may include accessing each entry of the surviving data matrix from the main memory at most once.

The processing core may include a plurality of processing cores. The computer instructions may be further configured to perform the step of scheduling the generating of the check data, the reconstructing of the lost original data, and the regenerating of the lost check data by: dividing the data matrix into a plurality of data matrices; dividing the surviving data matrix into a plurality of surviving data matrices; dividing the lost data matrix into a plurality of lost data matrices; dividing the check matrix into a plurality of check matrices; dividing the surviving check matrix into a plurality of surviving check matrices; dividing the lost check matrix into a plurality of lost check matrices; and assigning corresponding ones of the data matrices, the surviving data matrices, the lost data matrices, the check matrices, the surviving check matrices, and the lost check matrices to the processing cores to concurrently generate portions of the check data corresponding to the check matrices from respective ones of the data matrices, to concurrently reconstruct portions of the lost original data corresponding to the lost data matrices from respective ones of the surviving data matrices and the surviving check matrices, and to concurrently regenerate portions of the lost check data corresponding to the lost check matrices from respective ones of the surviving data matrices and respective portions of the reconstructed lost original data.

By providing practical and efficient systems and methods for erasure coding systems (which for byte-level processing can support up to $N+M=256$ drives, such as $N=127$ data drives and $M=129$ check drives, including a parity drive), applications such as RAID systems that can tolerate far more failing drives than was thought to be possible or practical can be implemented with accelerated performance significantly better than any prior art solution.

BRIEF DESCRIPTION OF THE DRAWINGS

The accompanying drawings, together with the specification, illustrate exemplary embodiments of the present inven-

tion and, together with the description, serve to explain aspects and principles of the present invention.

FIG. 1 shows an exemplary stripe of original and check data according to an embodiment of the present invention.

FIG. 2 shows an exemplary method for reconstructing lost data after a failure of one or more drives according to an embodiment of the present invention.

FIG. 3 shows an exemplary method for performing a parallel lookup Galois field multiplication according to an embodiment of the present invention.

FIG. 4 shows an exemplary method for sequencing the parallel lookup multiplier to perform the check data generation according to an embodiment of the present invention.

FIGS. 5-7 show an exemplary method for sequencing the parallel lookup multiplier to perform the lost data reconstruction according to an embodiment of the present invention.

FIG. 8 illustrates a multi-core architecture system according to an embodiment of the present invention.

FIG. 9 shows an exemplary disk drive configuration according to an embodiment of the present invention.

DETAILED DESCRIPTION

Hereinafter, exemplary embodiments of the invention will be described in more detail with reference to the accompanying drawings. In the drawings, like reference numerals refer to like elements throughout.

While optimal erasure codes have many applications, for ease of description, they will be described in this application with respect to RAID applications, i.e., erasure coding systems for the storage and retrieval of digital data distributed across numerous storage devices (or drives), though the present application is not limited thereto. For further ease of description, the storage devices will be assumed to be disk drives, though the invention is not limited thereto. In RAID systems, the data (or original data) is broken up into stripes, each of which includes N uniformly sized blocks (data blocks), and the N blocks are written across N separate drives (the data drives), one block per data drive.

In addition, for ease of description, blocks will be assumed to be composed of L elements, each element having a fixed size, say 8 bits or one byte. An element, such as a byte, forms the fundamental unit of operation for the RAID processing, but the invention is just as applicable to other size elements, such as 16 bits (2 bytes). For simplification, unless otherwise indicated, elements will be assumed to be one byte in size throughout the description that follows, and the term “element(s)” and “byte(s)” will be used synonymously.

Conceptually, different stripes can distribute their data blocks across different combinations of drives, or have different block sizes or numbers of blocks, etc., but for simplification and ease of description and implementation, the described embodiments in the present application assume a consistent block size (L bytes) and distribution of blocks among the data drives between stripes. Further, all variables, such as the number of data drives N, will be assumed to be positive integers unless otherwise specified. In addition, since the N=1 case reduces to simple data mirroring (that is, copying the same data drive multiple times), it will also be assumed for simplicity that N≥2 throughout.

The N data blocks from each stripe are combined using arithmetic operations (to be described in more detail below) in M different ways to produce M blocks of check data (check blocks), and the M check blocks written across M drives (the check drives) separate from the N data drives, one block per check drive. These combinations can take place, for example, when new (or changed) data is written to (or back to) disk. Accordingly, each of the N+M drives (data drives and check drives) stores a similar amount of data, namely one block for each stripe. As the processing of multiple stripes is concep-

tually similar to the processing of one stripe (only processing multiple blocks per drive instead of one), it will be further assumed for simplification that the data being stored or retrieved is only one stripe in size unless otherwise indicated. It will also be assumed that the block size L is sufficiently large that the data can be consistently divided across each block to produce subsets of the data that include respective portions of the blocks (for efficient concurrent processing by different processing units).

FIG. 1 shows an exemplary stripe 10 of original and check data according to an embodiment of the present invention.

Referring to FIG. 1, the stripe 10 can be thought of not only as the original N data blocks 20 that make up the original data, but also the corresponding M check blocks 30 generated from the original data (that is, the stripe 10 represents encoded data). Each of the N data blocks 20 is composed of L bytes 25 (labeled byte 1, byte 2, . . . , byte L), and each of the M check blocks 30 is composed of L bytes 35 (labeled similarly). In addition, check drive 1, byte 1, is a linear combination of data drive 1, byte 1; data drive 2, byte 1; . . . ; data drive N, byte 1. Likewise, check drive 1, byte 2, is generated from the same linear combination formula as check drive 1, byte 1, only using data drive 1, byte 2; data drive 2, byte 2; . . . ; data drive N, byte 2. In contrast, check drive 2, byte 1, uses a different linear combination formula than check drive 1, byte 1, but applies it to the same data, namely data drive 1, byte 1; data drive 2, byte 1; . . . ; data drive N, byte 1. In this fashion, each of the other check bytes 35 is a linear combination of the respective bytes of each of the N data drives 20 and using the corresponding linear combination formula for the particular check drive 30.

The stripe 10 in FIG. 1 can also be represented as a matrix C of encoded data. C has two sub-matrices, namely original data D on top and check data J on bottom. That is,

$$C = \begin{bmatrix} D \\ J \end{bmatrix} = \begin{bmatrix} D_{11} & D_{12} & \dots & D_{1L} \\ D_{21} & D_{22} & \dots & D_{2L} \\ \vdots & \vdots & \ddots & \vdots \\ D_{N1} & D_{N2} & \dots & D_{NL} \\ J_{11} & J_{12} & \dots & J_{1L} \\ J_{21} & J_{22} & \dots & J_{2L} \\ \vdots & \vdots & \ddots & \vdots \\ J_{M1} & J_{M2} & \dots & J_{ML} \end{bmatrix},$$

where D_{ij} =byte j from data drive i and J_{ij} =byte j from check drive i. Thus, the rows of encoded data C represent blocks, while the columns represent corresponding bytes of each of the drives.

Further, in case of a disk drive failure of one or more disks, the arithmetic operations are designed in such a fashion that for any stripe, the original data (and by extension, the check data) can be reconstructed from any combination of N data and check blocks from the corresponding N+M data and check blocks that comprise the stripe. Thus, RAID provides both parallel processing (reading and writing the data in stripes across multiple drives concurrently) and fault tolerance (regeneration of the original data even if as many as M of the drives fail), at the computational cost of generating the check data any time new data is written to disk, or changed data is written back to disk, as well as the computational cost of reconstructing any lost original data and regenerating any lost check data after a disk failure.

For example, for M=1 check drive, a single parity drive can function as the check drive (i.e., a RAID4 system). Here, the arithmetic operation is bitwise exclusive OR of each of the N corresponding data bytes in each data block of the stripe. In addition, as mentioned earlier, the assignment of parity

US 9,385,759 B2

11

blocks from different stripes to the same drive (i.e., RAID4) or different drives (i.e., RAID5) is arbitrary, but it does simplify the description and implementation to use a consistent assignment between stripes, so that will be assumed throughout. Since $M=1$ reduces to the case of a single parity drive, it will further be assumed for simplicity that $M \geq 2$ throughout.

For such larger values of M , Galois field arithmetic is used to manipulate the data, as described in more detail later. Galois field arithmetic, for Galois fields of powers-of-2 (such as 2^P) numbers of elements, includes two fundamental operations: (1) addition (which is just bitwise exclusive OR, as with the parity drive-only operations for $M=1$), and (2) multiplication. While Galois field (GF) addition is trivial on standard processors, GF multiplication is not. Accordingly, a significant component of RAID performance for $M \geq 2$ is speeding up the performance of GF multiplication, as will be discussed later. For purposes of description, GF addition will be represented by the symbol $+$ throughout while GF multiplication will be represented by the symbol \times throughout.

Briefly, in exemplary embodiments of the present invention, each of the M check drives holds linear combinations (over GF arithmetic) of the N data drives of original data, one linear combination (i.e., a GF sum of N terms, where each term represents a byte of original data times a corresponding factor (using GF multiplication) for the respective data drive) for each check drive, as applied to respective bytes in each block. One such linear combination can be a simple parity, i.e., entirely GF addition (all factors equal 1), such as a GF sum of the first byte in each block of original data as described above.

The remaining $M-1$ linear combinations include more involved calculations that include the nontrivial GF multiplication operations (e.g., performing a GF multiplication of the first byte in each block by a corresponding factor for the respective data drive, and then performing a GF sum of all these products). These linear combinations can be represented by an $(N+M) \times N$ matrix (encoding matrix or information dispersal matrix (IDM)) E of the different factors, one factor for each combination of (data or check) drive and data drive, with one row for each of the $N+M$ data and check drives and one column for each of the N data drives. The IDM E can also be represented as

$$\begin{bmatrix} I_N \\ H \end{bmatrix},$$

where I_N represents the $N \times N$ identity matrix (i.e., the original (unencoded) data) and H represents the $M \times N$ matrix of factors for the check drives (where each of the M rows corresponds to one of the M check drives and each of the N columns corresponds to one of the N data drives).

Thus,

$$E = \begin{bmatrix} I_N \\ H \end{bmatrix} = \begin{bmatrix} 1 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 1 \\ H_{11} & H_{12} & \dots & H_{1N} \\ H_{21} & H_{22} & \dots & H_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ H_{M1} & H_{M2} & \dots & H_{MN} \end{bmatrix},$$

12

where H_{ij} =factor for check drive i and data drive j . Thus, the rows of encoded data C represent blocks, while the columns represent corresponding bytes of each of the drives. In addition, check factors H , original data D , and check data J are related by the formula $J=H \times D$ (that is, matrix multiplication), or

$$\begin{bmatrix} J_{11} & J_{12} & \dots & J_{1L} \\ J_{21} & J_{22} & \dots & J_{2L} \\ \vdots & \vdots & \ddots & \vdots \\ J_{M1} & J_{M2} & \dots & J_{ML} \end{bmatrix} = \begin{bmatrix} H_{11} & H_{12} & \dots & H_{1N} \\ H_{21} & H_{22} & \dots & H_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ H_{M1} & H_{M2} & \dots & H_{MN} \end{bmatrix} \times \begin{bmatrix} D_{11} & D_{12} & \dots & D_{1L} \\ D_{21} & D_{22} & \dots & D_{2L} \\ \vdots & \vdots & \ddots & \vdots \\ D_{N1} & D_{N2} & \dots & D_{NL} \end{bmatrix},$$

where $J_{11} = (H_{11} \times D_{11}) + (H_{12} \times D_{21}) + \dots + (H_{1N} \times D_{N1})$,

$J_{12} = (H_{11} \times D_{12}) + (H_{12} \times D_{22}) + \dots + (H_{1N} \times D_{N2})$,

$J_{21} = (H_{21} \times D_{11}) + (H_{22} \times D_{21}) + \dots + (H_{2N} \times D_{N1})$,

and in general,

$J_{ij} = (H_{i1} \times D_{1j}) + (H_{i2} \times D_{2j}) + \dots + (H_{iN} \times D_{Nj})$

for $1 \leq i \leq M$

and $1 \leq j \leq L$.

Such an encoding matrix E is also referred to as an information dispersal matrix (IDM). It should be noted that matrices such as check drive encoding matrix H and identity matrix I_N also represent encoding matrices, in that they represent matrices of factors to produce linear combinations over GF arithmetic of the original data. In practice, the identity matrix I_N is trivial and may not need to be constructed as part of the IDM E . Only the encoding matrix E , however, will be referred to as the IDM. Methods of building an encoding matrix such as IDM E or check drive encoding matrix H are discussed below. In further embodiments of the present invention (as discussed further in Appendix A), such $(N+M) \times N$ (or $M \times N$) matrices can be trivially constructed (or simply indexed) from a master encoding matrix S , which is composed of $(N_{max} + M_{max}) \times N_{max}$ (or $M_{max} \times N_{max}$) bytes or elements, where $N_{max} + M_{max} = 256$ (or some other power of two) and $N \leq N_{max}$ and $M \leq M_{max}$. For example, one such master encoding matrix S can include a 127×127 element identity matrix on top (for up to $N_{max} = 127$ data drives), a row of 1's (for a parity drive), and a 128×127 element encoding matrix on bottom (for up to $M_{max} = 129$ check drives, including the parity drive), for a total of $N_{max} = 256$ drives.

The original data, in turn, can be represented by an $N \times L$ matrix D of bytes, each of the N rows representing the L bytes of a block of the corresponding one of the N data drives. If C represents the corresponding $(N+M) \times L$ matrix of encoded bytes (where each of the $N+M$ rows corresponds to one of the $N+M$ data and check drives), then C can be represented as $E \times$

$$D = \begin{bmatrix} I_N \\ H \end{bmatrix} \times D = \begin{bmatrix} I_N \times D \\ H \times D \end{bmatrix} = \begin{bmatrix} D \\ J \end{bmatrix},$$

where $J=H \times D$ is an $M \times L$ matrix of check data, with each of the M rows representing the L check bytes of the corresponding one of the M check drives. It should be noted that in the

relationships such as $C=E \times D$ or $J=H \times D$, \times represents matrix multiplication over the Galois field (i.e., GF multiplication and GF addition being used to generate each of the entries in, for example, C or J).

In exemplary embodiments of the present invention, the first row of the check drive encoding matrix H (or the $(N+1)^{th}$ row of the IDME) can be all 1's, representing the parity drive. For linear combinations involving this row, the GF multiplication can be bypassed and replaced with a GF sum of the corresponding bytes since the products are all trivial products involving the identity element 1. Accordingly, in parity drive implementations, the check drive encoding matrix H can also be thought of as an $(M-1) \times N$ matrix of non-trivial factors (that is, factors intended to be used in GF multiplication and not just GF addition).

Much of the RAID processing involves generating the check data when new or changed data is written to (or back to) disk. The other significant event for RAID processing is when one or more of the drives fail (data or check drives), or for whatever reason become unavailable. Assume that in such a failure scenario, F data drives fail and G check drives fail, where F and G are nonnegative integers. If $F=0$, then only check drives failed and all of the original data D survived. In this case, the lost check data can be regenerated from the original data D .

Accordingly, assume at least one data drive fails, that is, $F \geq 1$, and let $K=N-F$ represent the number of data drives that survive. K is also a nonnegative integer. In addition, let X represent the surviving original data and Y represent the lost original data. That is, X is a $K \times L$ matrix composed of the K rows of the original data matrix D corresponding to the K surviving data drives, while Y is an $F \times L$ matrix composed of the F rows of the original data matrix D corresponding to the F failed data drives.

$$\begin{bmatrix} X \\ Y \end{bmatrix}$$

thus represents a permuted original data matrix D' (that is, the original data matrix D , only with the surviving original data X on top and the lost original data Y on bottom. It should be noted that once the lost original data Y is reconstructed, it can be combined with the surviving original data X to restore the original data D , from which the check data for any of the failed check drives can be regenerated.

It should also be noted that $M-G$ check drives survive. In order to reconstruct the lost original data Y , enough (that is, at least IV) total drives must survive. Given that $K=N-F$ data drives survive, and that $M-G$ check drives survive, it follows that $(N-F)+(M-G) \geq N$ must be true to reconstruct the lost original data Y . This is equivalent to $F+G \leq M$ (i.e., no more than $F+G$ drives fail), or $F \leq M-G$ (that is, the number of failed data drives does not exceed the number of surviving check drives). It will therefore be assumed for simplicity that $F \leq M-G$.

In the routines that follow, performance can be enhanced by prebuilding lists of the failed and surviving data and check drives (that is, four separate lists). This allows processing of the different sets of surviving and failed drives to be done more efficiently than existing solutions, which use, for example, bit vectors that have to be examined one bit at a time and often include large numbers of consecutive zeros (or ones) when ones (or zeros) are the bit values of interest.

FIG. 2 shows an exemplary method 300 for reconstructing lost data after a failure of one or more drives according to an embodiment of the present invention.

While the recovery process is described in more detail later, briefly it consists of two parts: (1) determining the solution matrix, and (2) reconstructing the lost data from the surviving data. Determining the solution matrix can be done in three steps with the following algorithm (Algorithm 1), with reference to FIG. 2:

1. (Step 310 in FIG. 2) Reducing the $(M+N) \times N$ IDME to an $N \times N$ reduced encoding matrix T (also referred to as the transformed IDME) including the K surviving data drive rows and any F of the $M-G$ surviving check drive rows (for instance, the first F surviving check drive rows, as these will include the parity drive if it survived; recall that $F \leq M-G$ was assumed). In addition, the columns of the reduced encoding matrix T are rearranged so that the K columns corresponding to the K surviving data drives are on the left side of the matrix and the F columns corresponding to the F failed drives are on the right side of the matrix. (Step 320) These F surviving check drives selected to rebuild the lost original data Y will henceforth be referred to as "the F surviving check drives," and their check data W will be referred to as "the surviving check data," even though $M-G$ check drives survived. It should be noted that W is an $F \times L$ matrix composed of the F rows of the check data J corresponding to the F surviving check drives. Further, the surviving encoded data can be represented as a sub-matrix C' of the encoded data C . The surviving encoded data C' is an $N \times L$ matrix composed of the surviving original data X on top and the surviving check data W on bottom, that is,

$$C' = \begin{bmatrix} X \\ W \end{bmatrix}$$

2. (Step 330) Splitting the reduced encoding matrix T into four sub-matrices (that are also encoding matrices): (i) a $K \times K$ identity matrix I_K (corresponding to the K surviving data drives) in the upper left, (ii) a $K \times F$ matrix O of zeros in the upper right, (iii) an $F \times K$ encoding matrix A in the lower left corresponding to the F surviving check drive rows and the K surviving data drive columns, and (iv) an $F \times F$ encoding matrix B in the lower right corresponding to the F surviving check drive rows and the F failed data drive columns. Thus, the reduced encoding matrix T can be represented as

$$\begin{bmatrix} I_K & O \\ A & B \end{bmatrix}$$

3. (Step 340) Calculating the inverse B^{-1} of the $F \times F$ encoding matrix B . As is shown in more detail in Appendix A,

$$C' = T \times D', \text{ or } \begin{bmatrix} X \\ W \end{bmatrix} = \begin{bmatrix} I_K & O \\ A & B \end{bmatrix} \times \begin{bmatrix} X \\ Y \end{bmatrix},$$

which is mathematically equivalent to $W=A \times X+B \times Y$. B^{-1} is the solution matrix, and is itself an $F \times F$ encoding matrix. Calculating the solution matrix B^{-1} thus allows the lost origi-

US 9,385,759 B2

15

nal data Y to be reconstructed from the encoding matrices A and B along with the surviving original data X and the surviving check data W.

The $F \times K$ encoding matrix A represents the original encoding matrix E, only limited to the K surviving data drives and the F surviving check drives. That is, each of the F rows of A represents a different one of the F surviving check drives, while each of the K columns of A represents a different one of the K surviving data drives. Thus, A provides the encoding factors needed to encode the original data for the surviving check drives, but only applied to the surviving data drives (that is, the surviving partial check data). Since the surviving original data X is available, A can be used to generate this surviving partial check data.

In similar fashion, the $F \times F$ encoding matrix B represents the original encoding matrix E, only limited to the F surviving check drives and the F failed data drives. That is, the F rows of B correspond to the same F rows of A, while each of the F columns of B represents a different one of the F failed data drives. Thus, B provides the encoding factors needed to encode the original data for the surviving check drives, but only applied to the failed data drives (that is, the lost partial check data). Since the lost original data Y is not available, B cannot be used to generate any of the lost partial check data. However, this lost partial check data can be determined from A and the surviving check data W. Since this lost partial check data represents the result of applying B to the lost original data Y, B^{-1} thus represents the necessary factors to reconstruct the lost original data Y from the lost partial check data.

It should be noted that steps 1 and 2 in Algorithm 1 above are logical, in that encoding matrices A and B (or the reduced encoding matrix T, for that matter) do not have to actually be constructed. Appropriate indexing of the IDM E (or the master encoding matrix S) can be used to obtain any of their entries. Step 3, however, is a matrix inversion over GF arithmetic and takes $O(F^3)$ operations, as discussed in more detail later. Nonetheless, this is a significant improvement over existing solutions, which require $O(N^3)$ operations, since the number of failed data drives F is usually significantly less than the number of data drives N in any practical situation.

(Step 350 in FIG. 2) Once the encoding matrix A and the solution matrix B^{-1} are known, reconstructing the lost data from the surviving data (that is, the surviving original data X and the surviving check data W) can be accomplished in four steps using the following algorithm (Algorithm 2):

1. Use A and the surviving original data X (using matrix multiplication) to generate the surviving check data (i.e., $A \times X$), only limited to the K surviving data drives. Call this limited check data the surviving partial check data.
2. Subtract this surviving partial check data from the surviving check data W (using matrix subtraction, i.e., $W - A \times X$, which is just entry-by-entry GF subtraction, which is the same as GF addition for this Galois field). This generates the surviving check data, only this time limited to the F failed data drives. Call this limited check data the lost partial check data.
3. Use the solution matrix B^{-1} and the lost partial check data (using matrix multiplication, i.e., $B^{-1} \times (W - A \times X)$) to reconstruct the lost original data Y. Call this the recovered original data Y.
4. Use the corresponding rows of the IDM E (or master encoding matrix S) for each of the G failed check drives along with the original data D, as reconstructed from the surviving and recovered original data X and Y, to regenerate the lost check data (using matrix multiplication).

As will be shown in more detail later, steps 1-3 together require $O(F)$ operations times the amount of original data D to

16

reconstruct the lost original data Y for the F failed data drives (i.e., roughly 1 operation per failed data drive per byte of original data D), which is proportionally equivalent to the $O(M)$ operations times the amount of original data D needed to generate the check data J for the M check drives (i.e., roughly 1 operation per check drive per byte of original data D). In addition, this same equivalence extends to step 4, which takes $O(G)$ operations times the amount of original data D needed to regenerate the lost check data for the G failed check drives (i.e., roughly 1 operation per failed check drive per byte of original data D). In summary, the number of operations needed to reconstruct the lost data is $O(F+G)$ times the amount of original data D (i.e., roughly 1 operation per failed drive (data or check) per byte of original data D). Since $F+G \leq M$, this means that the computational complexity of Algorithm 2 (reconstructing the lost data from the surviving data) is no more than that of generating the check data J from the original data D.

As mentioned above, for exemplary purposes and ease of description, data is assumed to be organized in 8-bit bytes, each byte capable of taking on $2^8=256$ possible values. Such data can be manipulated in byte-size elements using GF arithmetic for a Galois field of size $2^8=256$ elements. It should also be noted that the same mathematical principles apply to any power-of-two 2^P number of elements, not just 256, as Galois fields can be constructed for any integral power of a prime number. Since Galois fields are finite, and since GF operations never overflow, all results are the same size as the inputs, for example, 8 bits.

In a Galois field of a power-of-two number of elements, addition and subtraction are the same operation, namely a bitwise exclusive OR (XOR) of the two operands. This is a very fast operation to perform on any current processor. It can also be performed on multiple bytes concurrently. Since the addition and subtraction operations take place, for example, on a byte-level basis, they can be done in parallel by using, for instance, x86 architecture Streaming SIMD Extensions (SSE) instructions (SIMD stands for single instruction, multiple data, and refers to performing the same instruction on different pieces of data, possibly concurrently), such as PXOR (Packed (bitwise) Exclusive OR).

SSE instructions can process, for example, 16-byte registers (XMM registers), and are able to process such registers as though they contain 16 separate one-byte operands (or 8 separate two-byte operands, or four separate four-byte operands, etc.) Accordingly, SSE instructions can do byte-level processing 16 times faster than when compared to processing a byte at a time. Further, there are 16 XMM registers, so dedicating four such registers for operand storage allows the data to be processed in 64-byte increments, using the other 12 registers for temporary storage. That is, individual operations can be performed as four consecutive SSE operations on the four respective registers (64 bytes), which can often allow such instructions to be efficiently pipelined and/or concurrently executed by the processor. In addition, the SSE instructions allows the same processing to be performed on different such 64-byte increments of data in parallel using different cores. Thus, using four separate cores can potentially speed up this processing by an additional factor of 4 over using a single core.

For example, a parallel adder (Parallel Adder) can be built using the 16-byte XMM registers and four consecutive PXOR instructions. Such parallel processing (that is, 64 bytes at a time with only a few machine-level instructions) for GF arithmetic is a significant improvement over doing the addition one byte at a time. Since the data is organized in blocks of any fixed number of bytes, such as 4096 bytes (4 kilobytes, or 4

US 9,385,759 B2

17

KB) or 32,768 bytes (32 KB), a block can be composed of numerous such 64-byte chunks (e.g., 64 separate 64-byte chunks in 4 KB, or 512 chunks in 32 KB).

Multiplication in a Galois field is not as straightforward. While much of it is bitwise shifts and exclusive OR's (i.e., "additions") that are very fast operations, the numbers "wrap" in peculiar ways when they are shifted outside of their normal bounds (because the field has only a finite set of elements), which can slow down the calculations. This "wrapping" in the GF multiplication can be addressed in many ways. For example, the multiplication can be implemented serially (Serial Multiplier) as a loop iterating over the bits of one operand while performing the shifts, adds, and wraps on the other operand. Such processing, however, takes several machine instructions per bit for 8 separate bits. In other words, this technique requires dozens of machine instructions per byte being multiplied. This is inefficient compared to, for example, the performance of the Parallel Adder described above.

For another approach (Serial Lookup Multiplier), multiplication tables (of all the possible products, or at least all the non-trivial products) can be pre-computed and built ahead of time. For example, a table of $256 \times 256 = 65,536$ bytes can hold all the possible products of the two different one-byte operands). However, such tables can force serialized access on what are only byte-level operations, and not take advantage of wide (concurrent) data paths available on modern processors, such as those used to implement the Parallel Adder above.

In still another approach (Parallel Multiplier), the GF multiplication can be done on multiple bytes at a time, since the same factor in the encoding matrix is multiplied with every element in a data block. Thus, the same factor can be multiplied with 64 consecutive data block bytes at a time. This is similar to the Parallel Adder described above, only there are several more operations needed to perform the operation. While this can be implemented as a loop on each bit of the factor, as described above, only performing the shifts, adds, and wraps on 64 bytes at a time, it can be more efficient to process the 256 possible factors as a (C language) switch statement, with inline code for each of 256 different combinations of two primitive GF operations: Multiply-by-2 and Add. For example, GF multiplication by the factor 3 can be effected by first doing a Multiply-by-2 followed by an Add. Likewise, GF multiplication by 4 is just a Multiply-by-2 followed by a Multiply-by-2 while multiplication by 6 is a Multiply-by-2 followed by an Add and then by another Multiply-by-2.

While this Add is identical to the Parallel Adder described above (e.g., four consecutive PXOR instructions to process 64 separate bytes), Multiply-by-2 is not as straightforward. For example, Multiply-by-2 in GF arithmetic can be implemented across 64 bytes at a time in 4 XMM registers via 4 consecutive PXOR instructions, 4 consecutive PCMPGTB (Packed Compare for Greater Than) instructions, 4 consecutive PADDB (Packed Add) instructions, 4 consecutive PAND (Bitwise AND) instructions, and 4 consecutive PXOR instructions. Though this takes 20 machine instructions, the instructions are very fast and results in 64 consecutive bytes of data at a time being multiplied by 2.

For 64 bytes of data, assuming a random factor between 0 and 255, the total overhead for the Parallel Multiplier is about 6 calls to multiply-by-2 and about 3.5 calls to add, or about $6 \times 20 + 3.5 \times 4 = 134$ machine instructions, or a little over 2 machine instructions per byte of data. While this compares favorably with byte-level processing, it is still possible to improve on this by building a parallel multiplier with a table lookup (Parallel Lookup Multiplier) using the PSHUFB

18

(Packed Shuffle Bytes) instruction and doing the GF multiplication in 4-bit nibbles (half bytes).

FIG. 3 shows an exemplary method 400 for performing a parallel lookup Galois field multiplication according to an embodiment of the present invention.

Referring to FIG. 3, in step 410, two lookup tables are built once: one lookup table for the low-order nibbles in each byte, and one lookup table for the high-order nibbles in each byte. Each lookup table contains 256 sets (one for each possible factor) of the 16 possible GF products of that factor and the 16 possible nibble values. Each lookup table is thus $256 \times 16 = 4096$ bytes, which is considerably smaller than the 65,536 bytes needed to store a complete one-byte multiplication table. In addition, PSHUFB does 16 separate table lookups at once, each for one nibble, so 8 PSHUFB instructions can be used to do all the table lookups for 64 bytes (128 nibbles).

Next, in step 420, the Parallel Lookup Multiplier is initialized for the next set of 64 bytes of operand data (such as original data or surviving original data). In order to save loading this data from memory on succeeding calls, the Parallel Lookup Multiplier dedicates four registers for this data, which are left intact upon exit of the Parallel Lookup Multiplier. This allows the same data to be called with different factors (such as processing the same data for another check drive).

Next in step 430, to process these 64 bytes of operand data, the Parallel Lookup Multiplier can be implemented with 2 MOVDQA (Move Double Quadword Aligned) instructions (from memory) to do the two table lookups and 4 MOVDQA instructions (register to register) to initialize registers (such as the output registers). These are followed in steps 440 and 450 by two nearly identical sets of 17 register-to-register instructions to carry out the multiplication 32 bytes at a time. Each such set starts (in step 440) with 5 more MOVDQA instructions for further initialization, followed by 2 PSRLW (Packed Shift Right Logical Word) instructions to realign the high-order nibbles for PSHUFB, and 4 PAND instructions to clear the high-order nibbles for PSHUFB. That is, two registers of byte operands are converted into four registers of nibble operands. Then, in step 450, 4 PSHUFB instructions are used to do the parallel table lookups, and 2 PXOR instructions to add the results of the multiplication on the two nibbles to the output registers.

Thus, the Parallel Lookup Multiplier uses 40 machine instructions to perform the parallel multiplication on 64 separate bytes, which is considerably better than the average 134 instructions for the Parallel Multiplier above, and only 10 times as many instructions as needed for the Parallel Adder. While some of the Parallel Lookup Multiplier's instructions are more complex than those of the Parallel Adder, much of this complexity can be concealed through the pipelined and/or concurrent execution of numerous such contiguous instructions (accessing different registers) on modern pipelined processors. For example, in exemplary implementations, the Parallel Lookup Multiplier has been timed at about 15 CPU clock cycles per 64 bytes processed per CPU core (about 0.36 clock cycles per instruction). In addition, the code footprint is practically nonexistent for the Parallel Lookup Multiplier (40 instructions) compared to that of the Parallel Multiplier (about 34,300 instructions), even when factoring the 8 KB needed for the two lookup tables in the Parallel Lookup Multiplier.

In addition, embodiments of the Parallel Lookup Multiplier can be passed 64 bytes of operand data (such as the next 64 bytes of surviving original data X to be processed) in four consecutive registers, whose contents can be preserved upon

exiting the Parallel Lookup Multiplier (and all in the same 40 machine instructions) such that the Parallel Lookup Multiplier can be invoked again on the same 64 bytes of data without having to access main memory to reload the data. Through such a protocol, memory accesses can be minimized (or significantly reduced) for accessing the original data D during check data generation or the surviving original data X during lost data reconstruction.

Further embodiments of the present invention are directed towards sequencing this parallel multiplication (and other GF) operations. While the Parallel Lookup Multiplier processes a GF multiplication of 64 bytes of contiguous data times a specified factor, the calls to the Parallel Lookup Multiplier should be appropriately sequenced to provide efficient processing. One such sequencer (Sequencer 1), for example, can generate the check data J from the original data D, and is described further with respect to FIG. 4.

The parity drive does not need GF multiplication. The check data for the parity drive can be obtained, for example, by adding corresponding 64-byte chunks for each of the data drives to perform the parity operation. The Parallel Adder can do this using 4 instructions for every 64 bytes of data for each of the N data drives, or N/16 instructions per byte.

The M-1 non-parity check drives can invoke the Parallel Lookup Multiplier on each 64-byte chunk, using the appropriate factor for the particular combination of data drive and check drive. One consideration is how to handle the data access. Two possible ways are:

- 1) "column-by-column," i.e., 64 bytes for one data drive, followed by the next 64 bytes for that data drive, etc., and adding the products to the running total in memory (using the Parallel Adder) before moving onto the next row (data drive); and
- 2) "row-by-row," i.e., 64 bytes for one data drive, followed by the corresponding 64 bytes for the next data drive, etc., and keeping a running total using the Parallel Adder, then moving onto the next set of 64-byte chunks.

Column-by-column can be thought of as "constant factor, varying data," in that the (GF multiplication) factor usually remains the same between iterations while the (64-byte) data changes with each iteration. Conversely, row-by-row can be thought of as "constant data, varying factor," in that the data usually remains the same between iterations while the factor changes with each iteration.

Another consideration is how to handle the check drives. Two possible ways are:

- a) one at a time, i.e., generate all the check data for one check drive before moving onto the next check drive; and
- b) all at once, i.e., for each 64-byte chunk of original data, do all of the processing for each of the check drives before moving onto the next chunk of original data.

While each of these techniques performs the same basic operations (e.g., 40 instructions for every 64 bytes of data for each of the N data drives and M-1 non-parity check drives, or $5N(M-1)/8$ instructions per byte for the Parallel Lookup Multiplier), empirical results show that combination (2)(b), that is, row-by-row data access on all of the check drives between data accesses performs best with the Parallel Lookup Multiplier. One reason may be that such an approach appears to minimize the number of memory accesses (namely, one) to each chunk of the original data D to generate the check data J. This embodiment of Sequencer 1 is described in more detail with reference to FIG. 4.

FIG. 4 shows an exemplary method 500 for sequencing the Parallel Lookup Multiplier to perform the check data generation according to an embodiment of the present invention.

Referring to FIG. 4, in step 510, the Sequencer 1 is called. Sequencer 1 is called to process multiple 64-byte chunks of data for each of the blocks across a stripe of data. For instance, Sequencer 1 could be called to process 512 bytes from each block. If, for example, the block size L is 4096 bytes, then it would take eight such calls to Sequencer 1 to process the entire stripe. The other such seven calls to Sequencer 1 could be to different processing cores, for instance, to carry out the check data generation in parallel. The number of 64-byte chunks to process at a time could depend on factors such as cache dimensions, input/output data structure sizes, etc.

In step 520, the outer loop processes the next 64-byte chunk of data for each of the drives. In order to minimize the number of accesses of each data drive's 64-byte chunk of data from memory, the data is loaded only once and preserved across calls to the Parallel Lookup Multiplier. The first data drive is handled specially since the check data has to be initialized for each check drive. Using the first data drive to initialize the check data saves doing the initialization as a separate step followed by updating it with the first data drive's data. In addition to the first data drive, the first check drive is also handled specially since it is a parity drive, so its check data can be initialized to the first data drive's data directly without needing the Parallel Lookup Multiplier.

In step 530, the first middle loop is called, in which the remainder of the check drives (that is, the non-parity check drives) have their check data initialized by the first data drive's data. In this case, there is a corresponding factor (that varies with each check drive) that needs to be multiplied with each of the first data drive's data bytes. This is handled by calling the Parallel Lookup Multiplier for each non-parity check drive.

In step 540, the second middle loop is called, which processes the other data drives' corresponding 64-byte chunks of data. As with the first data drive, each of the other data drives is processed separately, loading the respective 64 bytes of data into four registers (preserved across calls to the Parallel Lookup Multiplier). In addition, since the first check drive is the parity drive, its check data can be updated by directly adding these 64 bytes to it (using the Parallel Adder) before handling the non-parity check drives.

In step 550, the inner loop is called for the next data drive. In the inner loop (as with the first middle loop), each of the non-parity check drives is associated with a corresponding factor for the particular data drive. The factor is multiplied with each of the next data drive's data bytes using the Parallel Lookup Multiplier, and the results added to the check drive's check data.

Another such sequencer (Sequencer 2) can be used to reconstruct the lost data from the surviving data (using Algorithm 2). While the same column-by-column and row-by-row data access approaches are possible, as well as the same choices for handling the check drives, Algorithm 2 adds another dimension of complexity because of the four separate steps and whether to: (i) do the steps completely serially or (ii) do some of the steps concurrently on the same data. For example, step 1 (surviving check data generation) and step 4 (lost check data regeneration) can be done concurrently on the same data to reduce or minimize the number of surviving original data accesses from memory.

Empirical results show that method (2)(b)(ii), that is, row-by-row data access on all of the check drives and for both surviving check data generation and lost check data regeneration between data accesses performs best with the Parallel Lookup Multiplier when reconstructing lost data using Algorithm 2. Again, this may be due to the apparent minimization of the number of memory accesses (namely, one) of each

21

chunk of surviving original data X to reconstruct the lost data and the absence of memory accesses of reconstructed lost original data Y when regenerating the lost check data. This embodiment of Sequencer 1 is described in more detail with reference to FIGS. 5-7.

FIGS. 5-7 show an exemplary method 600 for sequencing the Parallel Lookup Multiplier to perform the lost data reconstruction according to an embodiment of the present invention.

Referring to FIG. 5, in step 610, the Sequencer 2 is called. Sequencer 2 has many similarities with the embodiment of Sequencer 1 illustrated in FIG. 4. For instance, Sequencer 2 processes the data drive data in 64-byte chunks like Sequencer 1. Sequencer 2 is more complex, however, in that only some of the data drive data is surviving; the rest has to be reconstructed. In addition, lost check data needs to be regenerated. Like Sequencer 1, Sequencer 2 does these operations in such a way as to minimize memory accesses of the data drive data (by loading the data once and calling the Parallel Lookup Multiplier multiple times). Assume for ease of description that there is at least one surviving data drive; the case of no surviving data drives is handled a little differently, but not significantly different. In addition, recall from above that the driving formula behind data reconstruction is $Y = B^{-1} \times (W - A \times X)$, where Y is the lost original data, B^{-1} is the solution matrix, W is the surviving check data, A is the partial check data encoding matrix (for the surviving check drives and the surviving data drives), and X is the surviving original data.

In step 620, the outer loop processes the next 64-byte chunk of data for each of the drives. Like Sequencer 1, the first surviving data drive is again handled specially since the partial check data $A \times X$ has to be initialized for each surviving check drive.

In step 630, the first middle loop is called, in which the partial check data $A \times X$ is initialized for each surviving check drive based on the first surviving data drive's 64 bytes of data. In this case, the Parallel Lookup Multiplier is called for each surviving check drive with the corresponding factor (from A) for the first surviving data drive.

In step 640, the second middle loop is called, in which the lost check data is initialized for each failed check drive. Using the same 64 bytes of the first surviving data drive (preserved across the calls to Parallel Lookup Multiplier in step 630), the Parallel Lookup Multiplier is again called, this time to initialize each of the failed check drive's check data to the corresponding component from the first surviving data drive. This completes the computations involving the first surviving data drive's 64 bytes of data, which were fetched with one access from main memory and preserved in the same four registers across steps 630 and 640.

Continuing with FIG. 6, in step 650, the third middle loop is called, which processes the other surviving data drives' corresponding 64-byte chunks of data. As with the first surviving data drive, each of the other surviving data drives is processed separately, loading the respective 64 bytes of data into four registers (preserved across calls to the Parallel Lookup Multiplier).

In step 660, the first inner loop is called, in which the partial check data $A \times X$ is updated for each surviving check drive based on the next surviving data drive's 64 bytes of data. In this case, the Parallel Lookup Multiplier is called for each surviving check drive with the corresponding factor (from A) for the next surviving data drive.

In step 670, the second inner loop is called, in which the lost check data is updated for each failed check drive. Using the same 64 bytes of the next surviving data drive (preserved

22

across the calls to Parallel Lookup Multiplier in step 660), the Parallel Lookup Multiplier is again called, this time to update each of the failed check drive's check data by the corresponding component from the next surviving data drive. This completes the computations involving the next surviving data drive's 64 bytes of data, which were fetched with one access from main memory and preserved in the same four registers across steps 660 and 670.

Next, in step 680, the computation of the partial check data $A \times X$ is complete, so the surviving check data W is added to this result (recall that $W - A \times X$ is equivalent to $W + A \times X$ in binary Galois Field arithmetic). This is done by the fourth middle loop, which for each surviving check drive adds the corresponding 64-byte component of surviving check data W to the (surviving) partial check data $A \times X$ (using the Parallel Adder) to produce the (lost) partial check data $W - A \times X$.

Continuing with FIG. 7, in step 690, the fifth middle loop is called, which performs the two dimensional matrix multiplication $B^{-1} \times (W - A \times X)$ to produce the lost original data Y . The calculation is performed one row at a time, for a total of F rows, initializing the row to the first term of the corresponding linear combination of the solution matrix B^{-1} and the lost partial check data $W - A \times X$ (using the Parallel Lookup Multiplier).

In step 700, the third inner loop is called, which completes the remaining $F-1$ terms of the corresponding linear combination (using the Parallel Lookup Multiplier on each term) from the fifth middle loop in step 690 and updates the running calculation (using the Parallel Adder) of the next row of $B^{-1} \times (W - A \times X)$. This completes the next row (and reconstructs the corresponding failed data drive's lost data) of lost original data Y , which can then be stored at an appropriate location.

In step 710, the fourth inner loop is called, in which the lost check data is updated for each failed check drive by the newly reconstructed lost data for the next failed data drive. Using the same 64 bytes of the next reconstructed lost data (preserved across calls to the Parallel Lookup Multiplier), the Parallel Lookup Multiplier is called to update each of the failed check drives' check data by the corresponding component from the next failed data drive. This completes the computations involving the next failed data drive's 64 bytes of reconstructed data, which were performed as soon as the data was reconstructed and without being stored and retrieved from main memory.

Finally, in step 720, the sixth middle loop is called. The lost check data has been regenerated, so in this step, the newly regenerated check data is stored at an appropriate location (if desired).

Aspects of the present invention can be also realized in other environments, such as two-byte quantities, each such two-byte quantity capable of taking on $2^{16} = 65,536$ possible values, by using similar constructs (scaled accordingly) to those presented here. Such extensions would be readily apparent to one of ordinary skill in the art, so their details will be omitted for brevity of description.

Exemplary techniques and methods for doing the Galois field manipulation and other mathematics behind RAID error correcting codes are described in Appendix A, which contains a paper "Information Dispersal Matrices for RAID Error Correcting Codes" prepared for the present application.

Multi-Core Considerations

What follows is an exemplary embodiment for optimizing or improving the performance of multi-core architecture systems when implementing the described erasure coding system routines. In multi-core architecture systems, each processor die is divided into multiple CPU cores, each with their

own local caches, together with a memory (bus) interface and possible on-die cache to interface with a shared memory with other processor dies.

FIG. 8 illustrates a multi-core architecture system **100** having two processor dies **110** (namely, Die **0** and Die **1**).

Referring to FIG. 8, each die **110** includes four central processing units (CPUs or cores) **120** each having a local level 1 (L1) cache. Each core **120** may have separate functional units, for example, an x86 execution unit (for traditional instructions) and a SSE execution unit (for software designed for the newer SSE instruction set). An example application of these function units is that the x86 execution unit can be used for the RAID control logic software while the SSE execution unit can be used for the GF operation software. Each die **110** also has a level 2 (L2) cache/memory bus interface **130** shared between the four cores **120**. Main memory **140**, in turn, is shared between the two dies **110**, and is connected to the input/output (I/O) controllers **150** that access external devices such as disk drives or other non-volatile storage devices via interfaces such as Peripheral Component Interconnect (PCI).

Redundant array of independent disks (RAID) controller processing can be described as a series of states or functions. These states may include: (1) Command Processing, to validate and schedule a host request (for example, to load or store data from disk storage); (2) Command Translation and Submission, to translate the host request into multiple disk requests and to pass the requests to the physical disks; (3) Error Correction, to generate check data and reconstruct lost data when some disks are not functioning correctly; and (4) Request Completion, to move data from internal buffers to requestor buffers. Note that the final state, Request Completion, may only be needed for a RAID controller that supports caching, and can be avoided in a cacheless design.

Parallelism is achieved in the embodiment of FIG. 8 by assigning different cores **120** to different tasks. For example, some of the cores **120** can be “command cores,” that is, assigned to the I/O operations, which includes reading and storing the data and check bytes to and from memory **140** and the disk drives via the I/O interface **150**. Others of the cores **120** can be “data cores,” and assigned to the GF operations, that is, generating the check data from the original data, reconstructing the lost data from the surviving data, etc., including the Parallel Lookup Multiplier and the sequencers described above. For example, in exemplary embodiments, a scheduler can be used to divide the original data **D** into corresponding portions of each block, which can then be processed independently by different cores **120** for applications such as check data generation and lost data reconstruction.

One of the benefits of this data core/command core subdivision of processing is ensuring that different code will be executed in different cores **120** (that is, command code in command cores, and data code in data cores). This improves the performance of the associated L1 cache in each core **120**, and avoids the “pollution” of these caches with code that is less frequently executed. In addition, empirical results show that the dies **110** perform best when only one core **120** on each die **110** does the GF operations (i.e., Sequencer 1 or Sequencer 2, with corresponding calls to Parallel Lookup Multiplier) and the other cores **120** do the I/O operations. This helps localize the Parallel Lookup Multiplier code and associated data to a single core **120** and not compete with other cores **120**, while allowing the other cores **120** to keep the data moving between memory **140** and the disk drives via the I/O interface **150**.

Embodiments of the present invention yield scalable, high performance RAID systems capable of outperforming other

systems, and at much lower cost, due to the use of high volume commodity components that are leveraged to achieve the result. This combination can be achieved by utilizing the mathematical techniques and code optimizations described elsewhere in this application with careful placement of the resulting code on specific processing cores. Embodiments can also be implemented on fewer resources, such as single-core dies and/or single-die systems, with decreased parallelism and performance optimization.

The process of subdividing and assigning individual cores **120** and/or dies **110** to inherently parallelizable tasks will result in a performance benefit. For example, on a Linux system, software may be organized into “threads,” and threads may be assigned to specific CPUs and memory systems via the `kthread_bind` function when the thread is created. Creating separate threads to process the GF arithmetic allows parallel computations to take place, which multiplies the performance of the system.

Further, creating multiple threads for command processing allows for fully overlapped execution of the command processing states. One way to accomplish this is to number each command, then use the arithmetic MOD function (% in C language) to choose a separate thread for each command. Another technique is to subdivide the data processing portion of each command into multiple components, and assign each component to a separate thread.

FIG. 9 shows an exemplary disk drive configuration **200** according to an embodiment of the present invention.

Referring to FIG. 9, eight disks are shown, though this number can vary in other embodiments. The disks are divided into three types: data drives **210**, parity drive **220**, and check drives **230**. The eight disks break down as three data drives **210**, one parity drive **220**, and four check drives **230** in the embodiment of FIG. 9.

Each of the data drives **210** is used to hold a portion of data. The data is distributed uniformly across the data drives **210** in stripes, such as 192 KB stripes. For example, the data for an application can be broken up into stripes of 192 KB, and each of the stripes in turn broken up into three 64 KB blocks, each of the three blocks being written to a different one of the three data drives **210**.

The parity drive **220** is a special type of check drive in that the encoding of its data is a simple summation (recall that this is exclusive OR in binary GF arithmetic) of the corresponding bytes of each of the three data drives **210**. That is, check data generation (Sequencer 1) or regeneration (Sequencer 2) can be performed for the parity drive **220** using the Parallel Adder (and not the Parallel Lookup Multiplier). Accordingly, the check data for the parity drive **220** is relatively straightforward to build. Likewise, when one of the data drives **210** no longer functions correctly, the parity drive **220** can be used to reconstruct the lost data by adding (same as subtracting in binary GF arithmetic) the corresponding bytes from each of the two remaining data drives **210**. Thus, a single drive failure of one of the data drives **210** is very straightforward to handle when the parity drive **220** is available (no Parallel Lookup Multiplier). Accordingly, the parity drive **220** can replace much of the GF multiplication operations with GF addition for both check data generation and lost data reconstruction.

Each of the check drives **230** contains a linear combination of the corresponding bytes of each of the data drives **210**. The linear combination is different for each check drive **230**, but in general is represented by a summation of different multiples of each of the corresponding bytes of the data drives **210** (again, all arithmetic being GF arithmetic). For example, for the first check drive **230**, each of the bytes of the first data drive **210** could be multiplied by 4, each of the bytes of the

US 9,385,759 B2

25

second data drive 210 by 3, and each of the bytes of the third data drive 210 by 6, then the corresponding products for each of the corresponding bytes could be added to produce the first check drive data. Similar linear combinations could be used to produce the check drive data for the other check drives 230. The specifics of which multiples for which check drive are explained in Appendix A.

With the addition of the parity drive 220 and check drives 230, eight drives are used in the RAID system 200 of FIG. 9. Accordingly, each 192 KB of original data is stored as 512 KB (i.e., eight blocks of 64 KB) of (original plus check) data. Such a system 200, however, is capable of recovering all of the original data provided any three of these eight drives survive. That is, the system 200 can withstand a concurrent failure of up to any five drives and still preserve all of the original data.

Exemplary Routines to Implement an Embodiment

The error correcting code (ECC) portion of an exemplary embodiment of the present invention may be written in software as, for example, four functions, which could be named as ECCInitialize, ECCSolve, ECCGenerate, and ECCRegenerate. The main functions that perform work are ECCGenerate and ECCRegenerate. ECCGenerate generates check codes for data that are used to recover data when a drive suffers an outage (that is, ECCGenerate generates the check data J from the original data D using Sequencer 1). ECCRegenerate uses these check codes and the remaining data to recover data after such an outage (that is, ECCRegenerate uses the surviving check data W, the surviving original data X, and Sequencer 2 to reconstruct the lost original data Y while also regenerating any of the lost check data). Prior to calling either of these functions, ECCSolve is called to compute the constants used for a particular configuration of data drives, check drives, and failed drives (for example, ECCSolve builds the solution matrix B^{-1} together with the lists of surviving and failed data and check drives). Prior to calling ECCSolve, ECCInitialize is called to generate constant tables used by all of the other functions (for example, ECCInitialize builds the IDM E and the two lookup tables for the Parallel Lookup Multiplier).

ECCInitialize

The function ECCInitialize creates constant tables that are used by all subsequent functions. It is called once at program initialization time. By copying or precomputing these values up front, these constant tables can be used to replace more time-consuming operations with simple table look-ups (such as for the Parallel Lookup Multiplier). For example, four tables useful for speeding up the GF arithmetic include:

1. mvct—an array of constants used to perform GF multiplication with the PSHUFB instruction that operates on SSE registers (that is, the Parallel Lookup Multiplier).
2. mast—contains the master encoding matrix S (or the Information Dispersal Matrix (IDM) E, as described in Appendix A), or at least the nontrivial portion, such as the check drive encoding matrix H
3. mul_tab—contains the results of all possible GF multiplication operations of any two operands (for example, $256 \times 256 = 65,536$ bytes for all of the possible products of two different one-byte quantities)
4. div_tab—contains the results of all possible GF division operations of any two operands (can be similar in size to mul_tab)

ECCSolve

The function ECCSolve creates constant tables that are used to compute a solution for a particular configuration of data drives, check drives, and failed drives. It is called prior to using the functions ECCGenerate or ECCRegenerate. It

26

allows the user to identify a particular case of failure by describing the logical configuration of data drives, check drives, and failed drives. It returns the constants, tables, and lists used to either generate check codes or regenerate data. For example, it can return the matrix B that needs to be inverted as well as the inverted matrix B^{-1} (i.e., the solution matrix).

ECCGenerate

The function ECCGenerate is used to generate check codes (that is, the check data matrix J) for a particular configuration of data drives and check drives, using Sequencer 1 and the Parallel Lookup Multiplier as described above. Prior to calling ECCGenerate, ECCSolve is called to compute the appropriate constants for the particular configuration of data drives and check drives, as well as the solution matrix B^{-1} .

ECCRegenerate

The function ECCRegenerate is used to regenerate data vectors and check code vectors for a particular configuration of data drives and check drives (that is, reconstructing the original data matrix D from the surviving data matrix X and the surviving check matrix W, as well as regenerating the lost check data from the restored original data), this time using Sequencer 2 and the Parallel Lookup Multiplier as described above. Prior to calling ECCRegenerate, ECCSolve is called to compute the appropriate constants for the particular configuration of data drives, check drives, and failed drives, as well as the solution matrix B^{-1} .

Exemplary Implementation Details

As discussed in Appendix A, there are two significant sources of computational overhead in erasure code processing (such as an erasure coding system used in RAID processing): the computation of the solution matrix B^{-1} for a given failure scenario, and the byte-level processing of encoding the check data J and reconstructing the lost data after a lost packet (e.g., data drive failure). By reducing the solution matrix B^{-1} to a matrix inversion of a $F \times F$ matrix, where F is the number of lost packets (e.g., failed drives), that portion of the computational overhead is for all intents and purposes negligible compared to the megabytes (MB), gigabytes (GB), and possibly terabytes (TB) of data that needs to be encoded into check data or reconstructed from the surviving original and check data. Accordingly, the remainder of this section will be devoted to the byte-level encoding and regenerating processing.

As already mentioned, certain practical simplifications can be assumed for most implementations. By using a Galois field of 256 entries, byte-level processing can be used for all of the GF arithmetic. Using the master encoding matrix S described in Appendix A, any combination of up to 127 data drives, 1 parity drive, and 128 check drives can be supported with such a Galois field. While, in general, any combination of data drives and check drives that adds up to 256 total drives is possible, not all combinations provide a parity drive when computed directly. Using the master encoding matrix S, on the other hand, allows all such combinations (including a parity drive) to be built (or simply indexed) from the same such matrix. That is, the appropriate sub-matrix (including the parity drive) can be used for configurations of less than the maximum number of drives.

In addition, using the master encoding matrix S permits further data drives and/or check drives can be added without requiring the recomputing of the IDM E (unlike other proposed solutions, which recompute E for every change of N or M). Rather, additional indexing of rows and/or columns of the master encoding matrix S will suffice. As discussed above, the use of the parity drive can eliminate or significantly reduce the somewhat complex GF multiplication operations

associated with the other check drives and replaces them with simple GF addition (bitwise exclusive OR in binary Galois fields) operations. It should be noted that master encoding matrices with the above properties are possible for any power-of-two number of drives $2^P=N_{max} M_{max}$ where the maximum number of data drives N_{max} is one less than a power of two (e.g., $N_{max}=127$ or 63) and the maximum number of check drives M_{max} (including the parity drive) is 2^P-N_{max} .

As discussed earlier, in an exemplary embodiment of the present invention, a modern x86 architecture is used (being readily available and inexpensive). In particular, this architecture supports 16 XMM registers and the SSE instructions. Each XMM register is 128 bits and is available for special purpose processing with the SSE instructions. Each of these XMM registers holds 16 bytes (8-bit), so four such registers can be used to store 64 bytes of data. Thus, by using SSE instructions (some of which work on different operand sizes, for example, treating each of the XMM registers as containing 16 one-byte operands), 64 bytes of data can be operated at a time using four consecutive SSE instructions (e.g., fetching from memory, storing into memory, zeroing, adding, multiplying), the remaining registers being used for intermediate results and temporary storage. With such an architecture, several routines are useful for optimizing the byte-level performance, including the Parallel Lookup Multiplier, Sequencer 1, and Sequencer 2 discussed above.

While the above description contains many specific embodiments of the invention, these should not be construed as limitations on the scope of the invention, but rather as examples of specific embodiments thereof. Accordingly, the scope of the invention should be determined not by the embodiments illustrated, but by the appended claims and their equivalents.

GLOSSARY OF SOME VARIABLES

A encoding matrix (F×K), sub-matrix of T
 B encoding matrix (F×F), sub-matrix of T
 B⁻¹ solution matrix (F×F)
 C encoded data matrix

$$((N + M) \times L) = \begin{bmatrix} D \\ J \end{bmatrix}$$

C' surviving encoded data matrix

$$(N \times L) = \begin{bmatrix} X \\ W \end{bmatrix}$$

D original data matrix (N×L)
 D' permuted original data matrix

$$(N \times L) = \begin{bmatrix} X \\ Y \end{bmatrix}$$

E information dispersal matrix

$$(IDM)((N + M) \times N) = \begin{bmatrix} I_N \\ H \end{bmatrix}$$

F number of failed data drives
 G number of failed check drives

H check drive encoding matrix (M×N)
 I identity matrix ($I_K=K \times K$ identity matrix, $I_N=N \times N$ identity matrix)
 J encoded check data matrix (M×L)
 K number of surviving data drives= $N-F$
 L data block size (elements or bytes)
 M number of check drives
 M_{max} maximum value of M
 N number of data drives
 N_{max} maximum value of N
 O zero matrix (K×F), sub-matrix of T
 S master encoding matrix ($(M_{max}+N_{max}) \times N_{max}$)
 T transformed IDM

$$(N \times N) = \begin{bmatrix} I_K & O \\ A & B \end{bmatrix}$$

W surviving check data matrix (F×L)
 X surviving original data matrix (K×L)
 Y lost original data matrix (F×L)

What is claimed is:

1. A system for accelerated error-correcting code (ECC) processing comprising:

- a processing core for executing computer instructions and accessing data from a main memory, the processing core comprising at least 16 data registers, each of the data registers comprising at least 16 bytes;
- one or more non-volatile storage media for storing the computer instructions and the data; and
- an input/output (I/O) controller for controlling data transfers between the main memory and the non-volatile storage media,

wherein the processing core, the non-volatile storage media, the I/O controller, and the computer instructions are configured to implement an erasure coding system comprising:

- a data matrix for holding original data in the main memory;
- a check matrix for holding check data in the main memory;
- an encoding matrix for holding first factors in the main memory, the first factors being for encoding the original data into the check data; and
- a thread for executing on the processing core and comprising:
 - a parallel multiplier for concurrently multiplying multiple data entries of a matrix by a single factor; and
 - a first sequencer for ordering data accesses through the data matrix and the encoding matrix using the parallel multiplier to generate the check data.

2. The system of claim 1, wherein the parallel multiplier is configured to process the data in units of at least 64 bytes spread over at least four of the data registers at a time.

3. The system of claim 2, wherein the parallel multiplier is further configured to:

- receive an input operand in the at least four of the data registers; and
- return with the input operand intact in the at least four of the data registers.

4. The system of claim 2, wherein consecutive ones of the computer instructions to process each of the units of the data access separate ones of the data registers to permit concurrent execution of the consecutive ones of the computer instructions on the processing core.

US 9,385,759 B2

29

5. The system of claim 1, wherein the parallel multiplier comprises two lookup tables for doing concurrent multiplication of 4-bit quantities across 16 byte-sized entries using the PSHUFB (Packed Shuffle Bytes) or equivalent instruction.

6. The system of claim 1, wherein the parallel multiplier is further configured to:

receive an input operand in at least one of the data registers; and

return with the input operand intact in the at least one of the data registers.

7. The system of claim 1, wherein the first sequencer is configured to access each entry of the data matrix from the main memory at most once while generating the check data.

8. A method of accelerated error-correcting code (ECC) processing on a computing system comprising a processing core for accessing instructions and data from a main memory, one or more non-volatile storage media for storing the instructions and the data, an input/output (I/O) controller for controlling data transfers between the main memory and the non-volatile storage media, and a computer program comprising a plurality of computer instructions for implementing an erasure coding system, the processing core comprising at least 16 data registers, each of the data registers comprising at least 16 bytes, the method comprising:

storing the computer program on the non-volatile storage media;

executing the computer instructions on the processing core;

transferring the data between the main memory and the non-volatile storage media using the I/O controller;

arranging original data as a data matrix in the main memory;

arranging first factors as an encoding matrix in the main memory, the first factors being for encoding the original data into check data, the check data being arranged as a check matrix in the main memory; and

generating the check data using a parallel multiplier for concurrently multiplying multiple data entries of a matrix by a single factor, the generating of the check data comprising ordering data accesses through the data matrix and the encoding matrix using the parallel multiplier.

9. The method of claim 8, wherein the generating of the check data further comprises processing the data by the parallel multiplier in units of at least 64 bytes spread over at least four of the data registers at a time.

10. The method of claim 9, wherein the generating of the check data further comprises:

receiving by the parallel multiplier an input operand in the at least four of the data registers; and

returning by the parallel multiplier the input operand intact in the at least four of the data registers.

11. The method of claim 9, wherein

consecutive ones of the computer instructions that process each of the units of the data access separate ones of the data registers,

the executing of the computer instructions on the processing core further comprises concurrently executing the consecutive ones of the computer instructions on the processing core.

12. The method of claim 8, wherein the parallel multiplier comprises two lookup tables and the generating of the check data further comprises using the parallel multiplier with the two lookup tables to do concurrent multiplication of 4-bit

30

quantities across 16 byte-sized entries using the PSHUFB (Packed Shuffle Bytes) or equivalent instruction.

13. The method of claim 8, wherein the generating of the check data further comprises:

receiving by the parallel multiplier an input operand in at least one of the data registers; and

returning by the parallel multiplier the input operand intact in the at least one of the data registers.

14. The method of claim 8, wherein the generating of the check data comprises accessing each entry of the data matrix from the main memory at most once.

15. A non-transitory computer-readable storage medium containing a computer program comprising a plurality of computer instructions for performing accelerated error-correcting code (ECC) processing on a computing system comprising a processing core for accessing instructions and data from a main memory, the processing core comprising at least 16 data registers, each of the data registers comprising at least 16 bytes, the computer instructions being configured to implement an erasure coding system when executed on the computing system by performing the steps of:

arranging original data as a data matrix in the main memory;

arranging first factors as an encoding matrix in the main memory, the first factors being for encoding the original data into check data, the check data being arranged as a check matrix in the main memory; and

generating the check data using a parallel multiplier for concurrently multiplying multiple data entries of a matrix by a single factor, the generating of the check data comprising ordering data accesses through the data matrix and the encoding matrix using the parallel multiplier.

16. The storage medium of claim 15, wherein the generating of the check data further comprises processing the data by the parallel multiplier in units of at least 64 bytes spread over at least four of the data registers at a time.

17. The storage medium of claim 16, wherein the generating of the check data further comprises:

receiving by the parallel multiplier an input operand in the at least four of the data registers; and

returning by the parallel multiplier the input operand intact in the at least four of the data registers.

18. The storage medium of claim 16, wherein consecutive ones of the computer instructions that process each of the units of the data access separate ones of the data registers,

the executing of the computer instructions on the processing core further comprises concurrently executing the consecutive ones of the computer instructions on the processing core.

19. The storage medium of claim 15, wherein the parallel multiplier comprises two lookup tables and the generating of the check data further comprises using the parallel multiplier with the two lookup tables to do concurrent multiplication of 4-bit quantities across 16 byte-sized entries using the PSHUFB (Packed Shuffle Bytes) or equivalent instruction.

20. The storage medium of claim 15, wherein the generating of the check data further comprises:

receiving by the parallel multiplier an input operand in at least one of the data registers; and

returning by the parallel multiplier the input operand intact in the at least one of the data registers.

* * * * *

EXHIBIT D



US010003358B2

(12) **United States Patent**
Anderson

(10) **Patent No.:** **US 10,003,358 B2**
(45) **Date of Patent:** ***Jun. 19, 2018**

(54) **ACCELERATED ERASURE CODING SYSTEM AND METHOD**

(56) **References Cited**

(71) Applicant: **STREAMSCALE, INC.**, Los Angeles, CA (US)

5,577,054 A 11/1996 Pharris
5,754,563 A 5/1998 White

(72) Inventor: **Michael H. Anderson**, Los Angeles, CA (US)

(Continued)

(73) Assignee: **StreamScale, Inc.**, Los Angeles, CA (US)

OTHER PUBLICATIONS

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 0 days. days.

M. Lalam, et al. "Sliding Encoding-Window for Reed-Solomon code decoding," 4th International Symposium on Turbo Codes & Related Topics; 6th International ITG-Conference on Source and Channel Coding, Munich, Germany, 2006, pp. 1-6.

This patent is subject to a terminal disclaimer.

(Continued)

Primary Examiner — John J Tabone, Jr.

(21) Appl. No.: **15/201,196**

(74) *Attorney, Agent, or Firm* — Lewis Roca Rothgerber Christie LLP

(22) Filed: **Jul. 1, 2016**

(65) **Prior Publication Data**

(57) **ABSTRACT**

US 2017/0005671 A1 Jan. 5, 2017

Related U.S. Application Data

An accelerated erasure coding system includes a processing core for executing computer instructions and accessing data from a main memory, and a non-volatile storage medium for storing the computer instructions. The processing core, storage medium, and computer instructions are configured to implement an erasure coding system, which includes: a data matrix for holding original data in the main memory; a check matrix for holding check data in the main memory; an encoding matrix for holding first factors in the main memory, the first factors being for encoding the original data into the check data; and a thread for executing on the processing core. The thread includes: a parallel multiplier for concurrently multiplying multiple entries of the data matrix by a single entry of the encoding matrix; and a first sequencer for ordering operations through the data matrix and the encoding matrix using the parallel multiplier to generate the check data.

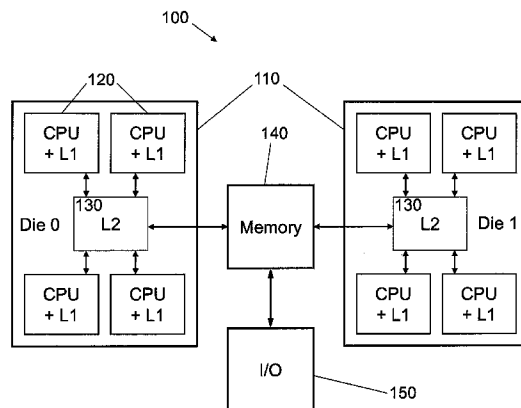
(63) Continuation of application No. 14/852,438, filed on Sep. 11, 2015, now Pat. No. 9,385,759, which is a (Continued)

(51) **Int. Cl.**
H03M 13/15 (2006.01)
G06F 11/10 (2006.01)
(Continued)

(52) **U.S. Cl.**
CPC **H03M 13/154** (2013.01); **G06F 11/1068** (2013.01); **G06F 11/1076** (2013.01);
(Continued)

(58) **Field of Classification Search**
CPC H03M 13/373; H03M 13/3761; H03M 13/3776; H03M 13/616; H03M 13/1191;
(Continued)

57 Claims, 9 Drawing Sheets



US 10,003,358 B2

Page 2

Related U.S. Application Data

continuation of application No. 14/223,740, filed on Mar. 24, 2014, now Pat. No. 9,160,374, which is a continuation of application No. 13/341,833, filed on Dec. 30, 2011, now Pat. No. 8,683,296.

(51) Int. Cl.

H03M 13/11 (2006.01)
H03M 13/13 (2006.01)
G06F 12/02 (2006.01)
G06F 12/06 (2006.01)
H03M 13/37 (2006.01)
H03M 13/00 (2006.01)
H04L 1/00 (2006.01)
G11C 29/52 (2006.01)

(52) U.S. Cl.

CPC **G06F 11/1092** (2013.01); **G06F 11/1096** (2013.01); **G06F 12/0238** (2013.01); **G06F 12/06** (2013.01); **G11C 29/52** (2013.01); **H03M 13/1191** (2013.01); **H03M 13/134** (2013.01); **H03M 13/1515** (2013.01); **H03M 13/373** (2013.01); **H03M 13/3761** (2013.01); **H03M 13/3776** (2013.01); **H03M 13/616** (2013.01); **H03M 13/6502** (2013.01); **H04L 1/0043** (2013.01); **H04L 1/0057** (2013.01); **G06F 2211/109** (2013.01); **G06F 2211/1057** (2013.01)

(58) Field of Classification Search

CPC H03M 13/134; H03M 13/1515; H03M 13/154; H03M 13/6502; H04L 1/0043; H04L 1/0057; G06F 11/1076; G06F 11/1092; G06F 11/1096; G06F 12/0238; G06F 12/06; G06F 2211/1057; G06F 2211/109; G06F 11/1068; G11C 29/52
 USPC 714/6.24, 6.1, 6.11, 6.2, 6.21, 6.32, 763, 714/752, 758, 768, 770, 773, 784, 786
 See application file for complete search history.

(56)

References Cited

U.S. PATENT DOCUMENTS

6,486,803 B1 11/2002 Luby et al.
 6,654,924 B1* 11/2003 Hassner G11B 20/1813
 714/758
 6,823,425 B2* 11/2004 Ghosh G06F 11/1076
 711/114
 7,350,126 B2* 3/2008 Winograd G06F 11/1076
 714/752
 7,865,809 B1 1/2011 Lee et al.
 7,930,337 B2 4/2011 Hasenplaugh et al.
 8,145,941 B2* 3/2012 Jacobson G06F 11/1076
 714/6.24
 8,352,847 B2* 1/2013 Gunnam G06F 17/16
 714/758
 8,683,296 B2* 3/2014 Anderson H03M 13/1515
 714/6.24
 8,914,706 B2* 12/2014 Anderson G06F 11/1076
 714/6.24
 9,160,374 B2* 10/2015 Anderson H03M 13/134
 9,385,759 B2* 7/2016 Anderson H03M 13/373
 2009/0055717 A1 2/2009 Au et al.
 2009/0249170 A1 10/2009 Maiuzzo
 2010/0293439 A1 11/2010 Flynn et al.
 2011/0029756 A1* 2/2011 Biscondi H03M 13/1114
 712/22
 2012/0272036 A1* 10/2012 Muralimanohar .. G06F 12/0238
 711/202

2013/0108048 A1* 5/2013 Grube H04W 12/00
 380/270
 2013/0110962 A1* 5/2013 Grube H04W 12/00
 709/213
 2013/0111552 A1* 5/2013 Grube H04W 12/00
 726/3
 2013/0124932 A1* 5/2013 Schuh G06F 9/44
 714/718
 2013/0173956 A1* 7/2013 Anderson G06F 11/1076
 714/6.24
 2013/0173996 A1* 7/2013 Anderson H03M 13/134
 714/770
 2014/0040708 A1 2/2014 Maiuzzo
 2014/0068391 A1 3/2014 Goel et al.
 2015/0012796 A1* 1/2015 Anderson H03M 13/134
 714/763

OTHER PUBLICATIONS

Neifeld, M.A & Sridharan, S. K. (1997). Parallel error correction using spectral Reed-Solomon codes. *Journal of Optical Communications*, 18(4), pp. 144-150.
 Casey Henderson: Letter to the USENIX Community <https://www.usenix.org/system/files/conference/fast13/fast13_memo_021715.pdf> Feb. 17, 2015.
 Chandan Kumar Singh: EC Jerasure plugin and StreamScale Inc, <<http://www.spinics.net/lists/ceph-devel/msg29944.html>> Apr. 20, 2016.
 Code Poetry and Text Adventures: <<http://catid.mechafetus.com/news/news.php?view=381>> Dec. 14, 2014.
 Curtis Chan: StreamScale Announces Settlement of Erasure Code Technology Patent Litigation, <<http://www.prweb.com/releases/2014/12/prweb12368357.htm>>, Dec. 3, 2014.
 Ethan Miller, <<https://plus.google.com/113956021908222328905/posts/bPcYevPkJWd>>, Aug. 2, 2013.
 H. Peter Anvin. "The mathematics of RAID-6." 2004, 2011.
 Hafner et al., Matrix Methods for Lost Data Reconstruction in Erasure Codes, Nov. 16, 2005, USENIX FAST '05 Paper, pp. 1-26.
 James S. Plank, Ethan L. Miller, Will B. Houston: GF-Complete: A Comprehensive Open Source Library for Galois Field Arithmetic, <<http://web.eecs.utk.edu/~plank/plank/papers/CS-13-703.html>> Jan. 2013.
 James S. Plank, Jianqiang Luo, Catherine D. Schuman, Lihao Xu, Zooko Wilcox-O'Hearn: A Performance Evaluation and Examination of Open-Source Erasure Coding Libraries for Storage, <https://www.usenix.org/legacy/event/fast09/tech/full_papers/plank/plank_html/_22_2009>.
 Kevin M. Greenan, Ethan L. Miller, Thomas J.E. Schwarz, S. J.: Optimizing Galois Field Arithmetic for Diverse Processor Architectures and Applications, *Proceedings of the 16th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS 2008)*, Baltimore, MD, Sep. 2008.
 Lee, "High-Speed VLSI Architecture for Parallel Reed-Solomon Decoder", *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 11, No. 2, Apr. 2003, pp. 288-294.
 Li et al.; Preventing Silent Data Corruptions from Propagating During Data Reconstruction; *IEEE Transactions on Computers*, vol. 59, No. 12, Dec. 2010; pp. 1611-1624.
 Li Han and Qian Huan-yan. "Parallelized Network Coding with SIMD instruction sets." In *Computer Science and Computational Technology, 2008. ISCSCT'08. International Symposium on*, vol. 1, pp. 364-369. IEEE, 2008.
 Loic Dachary: Deadline of Github pull request for Hammer release, <<http://www.spinics.net/lists/ceph-devel/msg22001.html>> Jan. 13, 2015.
 Louis Lavile: <<https://twitter.com/louislavile>> Nov. 13, 2014.
 Maddock, et al.; White Paper, Surviving Two Disk Failures Introducing Various "RAID 6" Implementations; Xyratex; pp. 1-13.
 Mann, "The Original View of Reed-Solomon Coding and the Welch-Berlekamp Decoding Algorithm", A Dissertation Submitted to the Faculty of the Graduate Interdisciplinary Program in Applied Mathematics, The University of Arizona, Jul. 19, 2013, 143 sheets.

US 10,003,358 B2

Page 3

(56)

References Cited

OTHER PUBLICATIONS

Marius Gedminas: <<http://eavesdrop.openstack.org/irclogs/%23openstack-swift/%23openstack-swift.2015-04-30.log.html>> Apr. 29, 2015.

Matthew L. Curry, Anthony Skjellum, H. Lee Ward, and Ron Brightwell. "Arbitrary dimension reed-solomon coding and decoding for extended raid on gpus." In *Petascale Data Storage Workshop, 2008. PDSW'08. 3rd*, pp. 1-3. IEEE, 2008.

Matthew L. Curry, Anthony Skjellum, H. Lee Ward, Ron Brightwell: Gibraltar: A Reed-Solomon coding library for storage applications on programmable graphics processors. *Concurrency and Computation: Practice and Experience* 23(18): pp. 2477-2495 (2011).

Matthew L. Curry, H. Lee Ward, Anthony Skjellum, Ron Brightwell: A Lightweight, GPU-Based Software RAID System. *ICPP 2010*: pp. 565-572.

Matthew L. Curry, Lee H. Ward, Anthony Skjellum, and Ron B. Brightwell: Accelerating Reed-Solomon Coding in RAID Systems With GPUs, *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*. IEEE, 2008.

Michael A. O'Shea: StreamScale, <<https://lists.ubuntu.com/archives/technical-board/2015-April/002100.html>> Apr. 29, 2015.

Mike Masnik: Patent Troll Kills Open Source Project on Speeding Up the Computation of Erasure Codes, <<https://www.techdirt.com/>

[articles/20141115/07113529155/patent-troll-kills-open-source-project-speeding-up-computation-erasure-codes.shtml](https://www.techdirt.com/articles/20141115/07113529155/patent-troll-kills-open-source-project-speeding-up-computation-erasure-codes.shtml)>, Nov. 19, 2014.

Plank; All About Erasure Codes:—Reed-Solomon Coding—LDPC Coding; Logistical Computing and Internetworking Laboratory, Department of Computer Science, University of Tennessee; ICL—Aug. 20, 2004; 52 sheets.

Robert Louis Cloud, Matthew L. Curry, H. Lee Ward, Anthony Skjellum, Purushotham Bangalore: Accelerating Lossless Data Compression with GPUs. CoRR abs/1107.1525 (2011).

Roy Schestowitz: US Patent Reform (on Trolls Only) More or Less Buried or Ineffective, <<http://techrights.org/2014/12/12/us-patent-reform/>> Dec. 12, 2014.

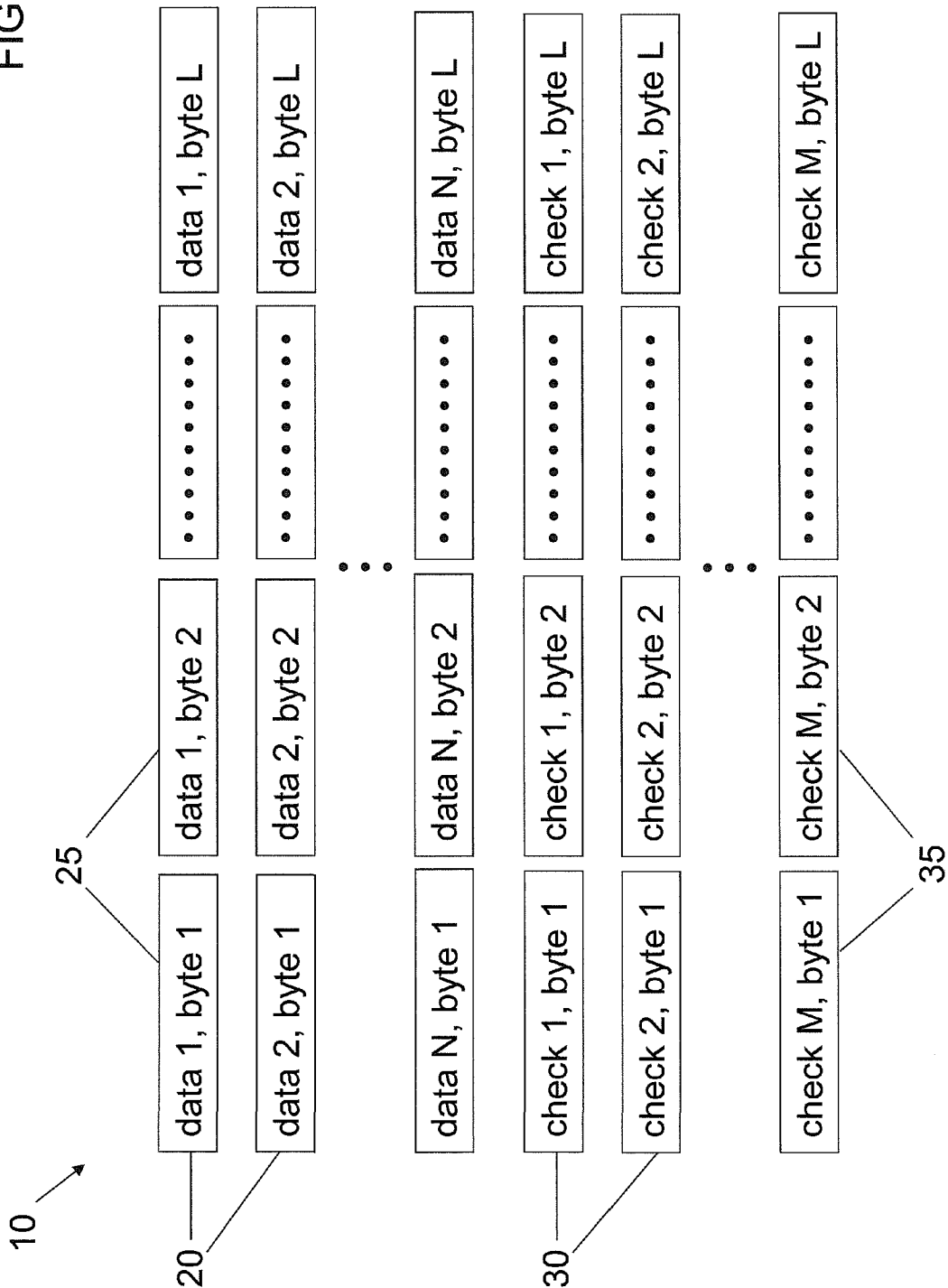
Weibin Sun, Robert Ricci, Matthew L. Curry: GPUstore: harnessing GPU computing for storage systems in the OS kernel. *SYSTOR 2012*: p. 6.

Xin Zhou, Anthony Skjellum, Matthew L. Curry: Abstract: Extended Abstract for Evaluating Asynchrony in Gibraltar RAID's GPU Reed-Solomon Coding Library. *SC Companion 2012*: pp. 1496-1497.

Xin Zhou, Anthony Skjellum, Matthew L. Curry: Poster: Evaluating Asynchrony in Gibraltar RAID's GPU Reed-Solomon Coding Library. *SC Companion 2012*: p. 1498.

* cited by examiner

FIG. 1



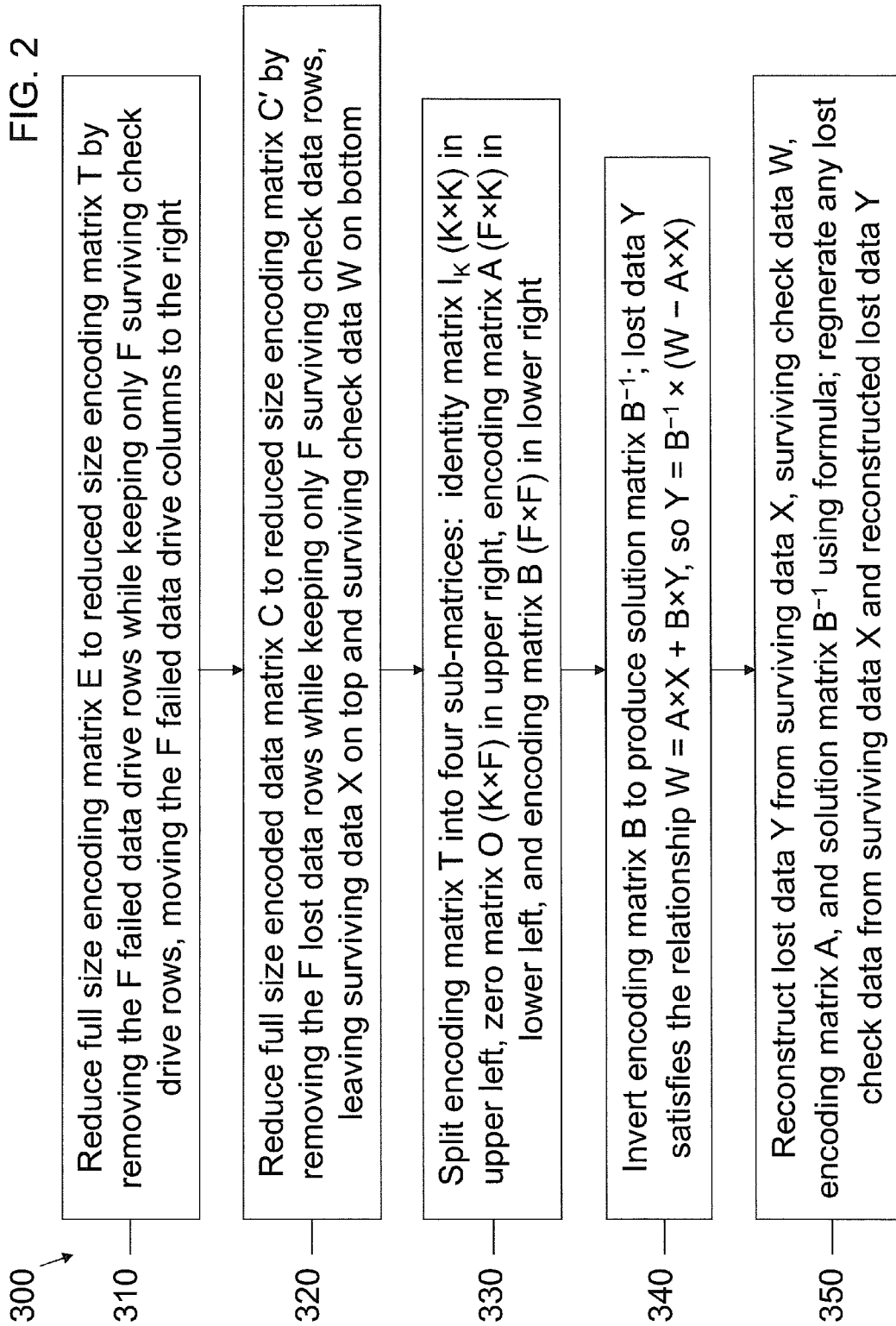


FIG. 3

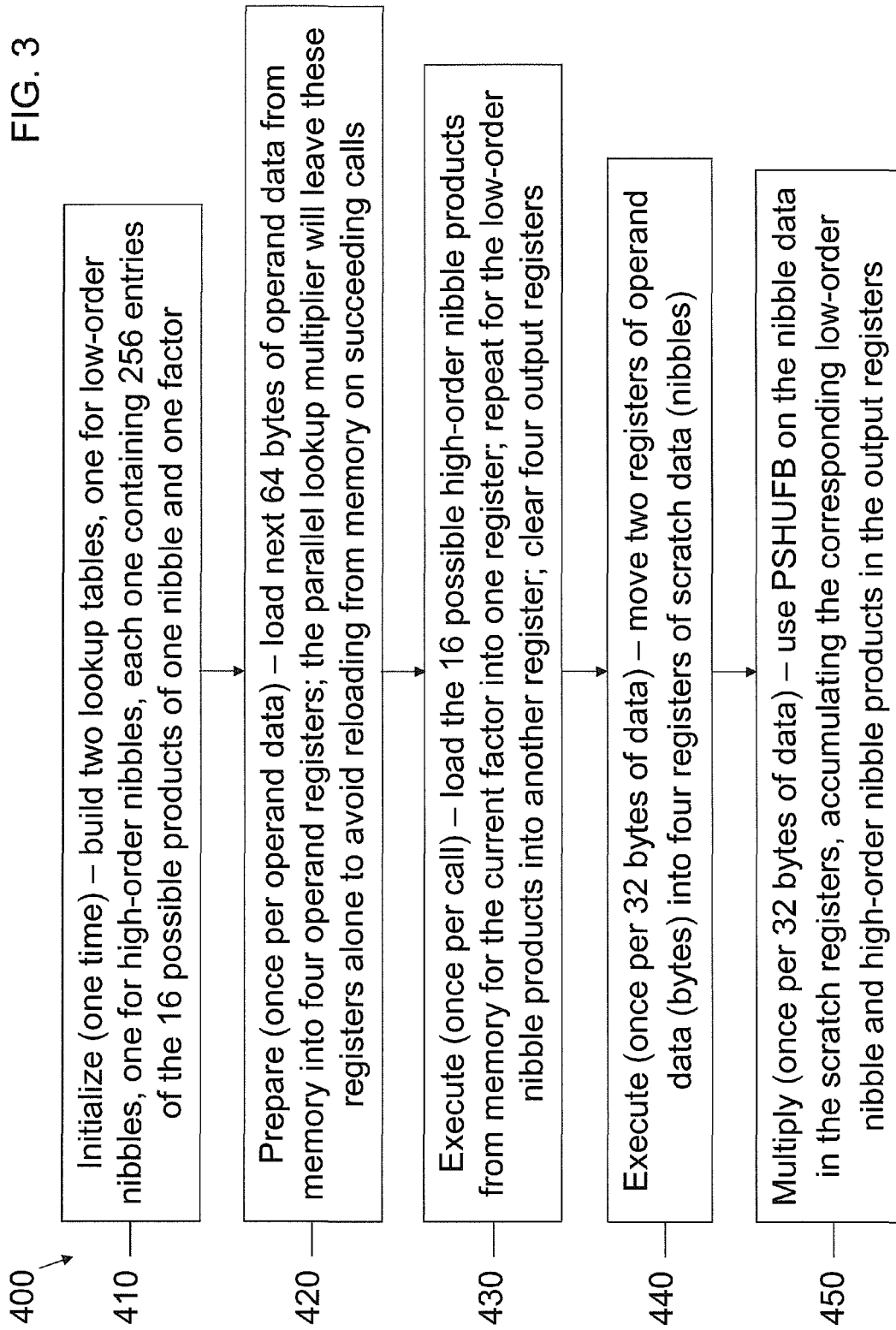


FIG. 4

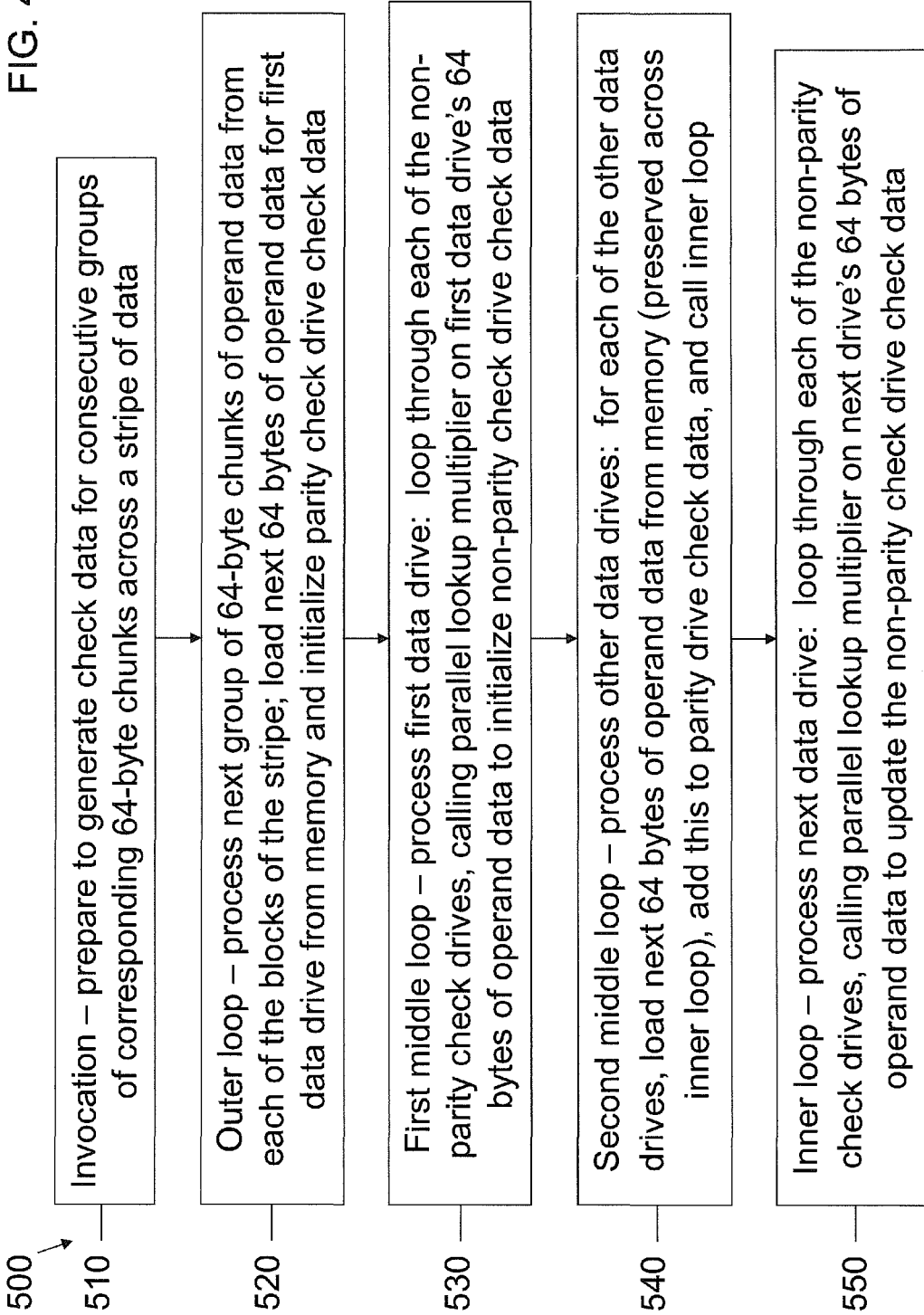


FIG. 5

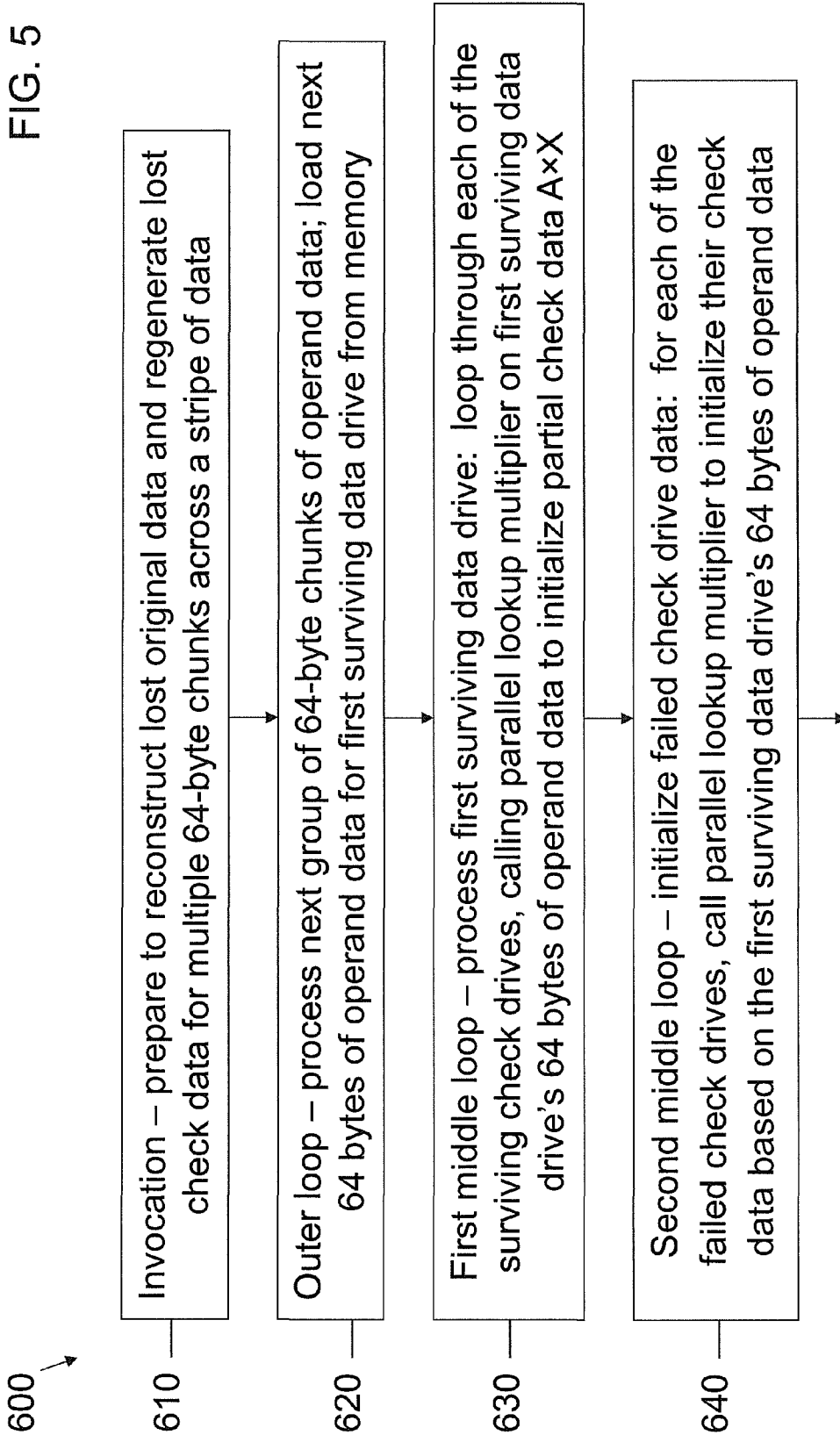


FIG. 6

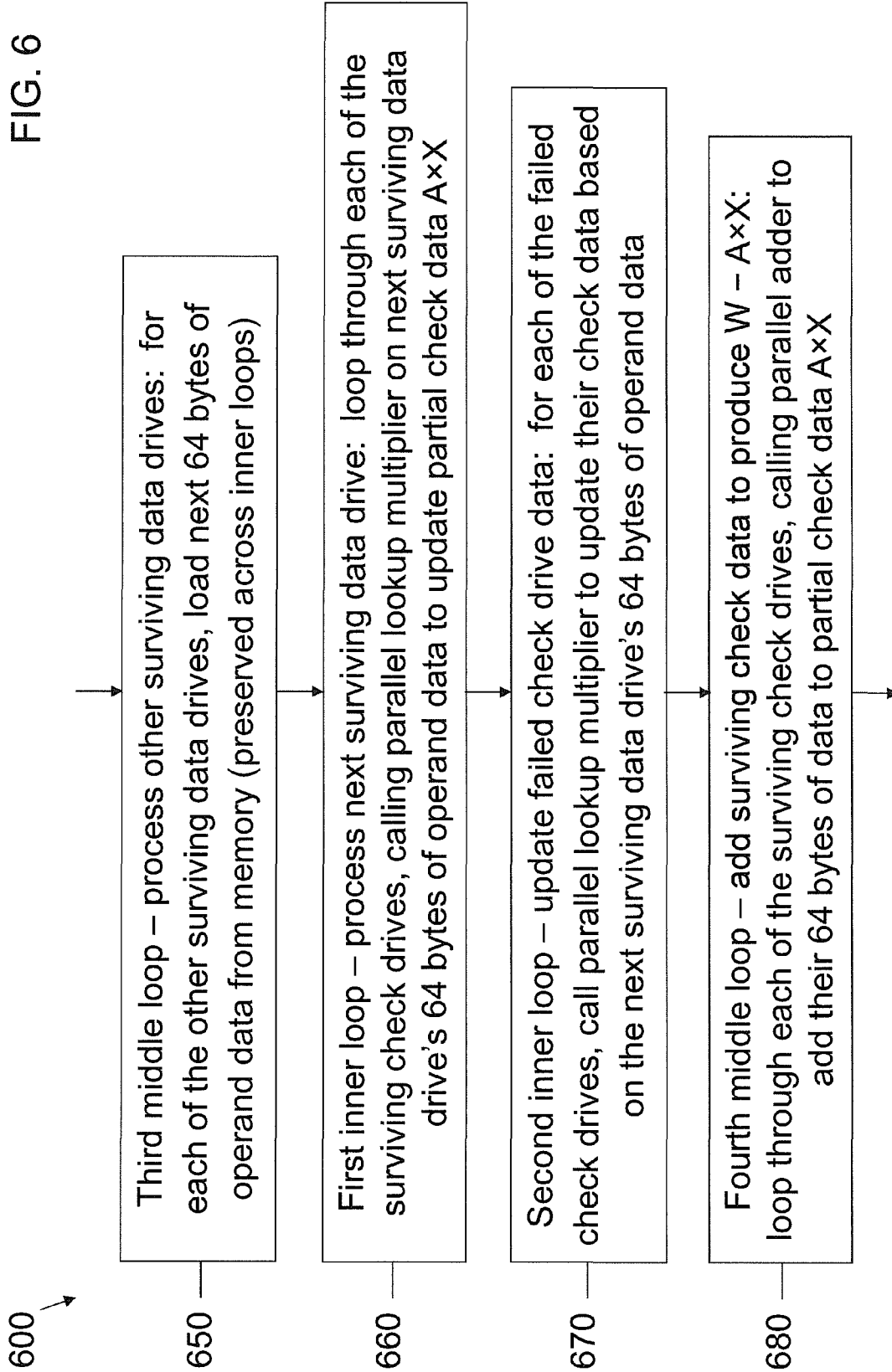


FIG. 7

600 ↗

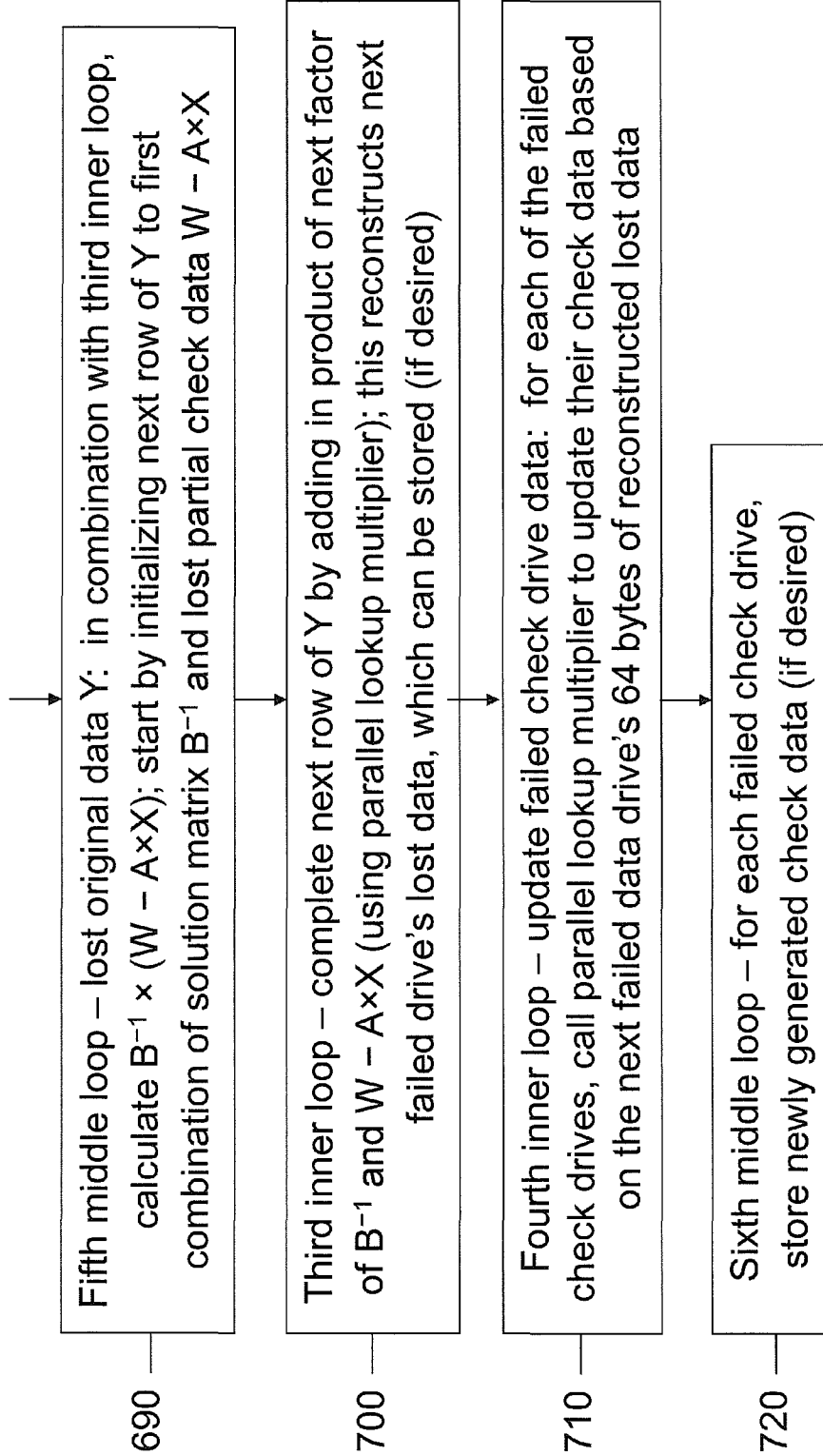


FIG. 8

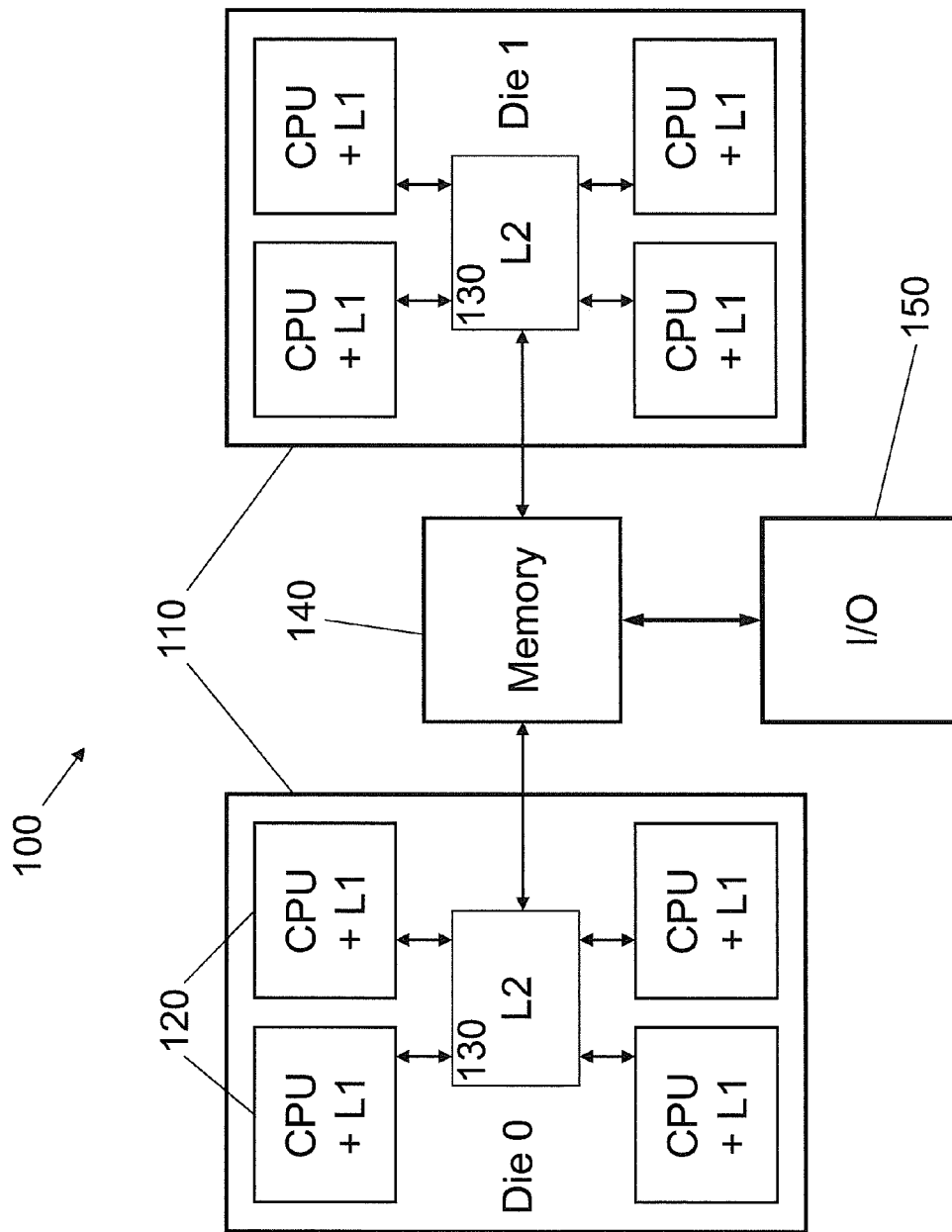
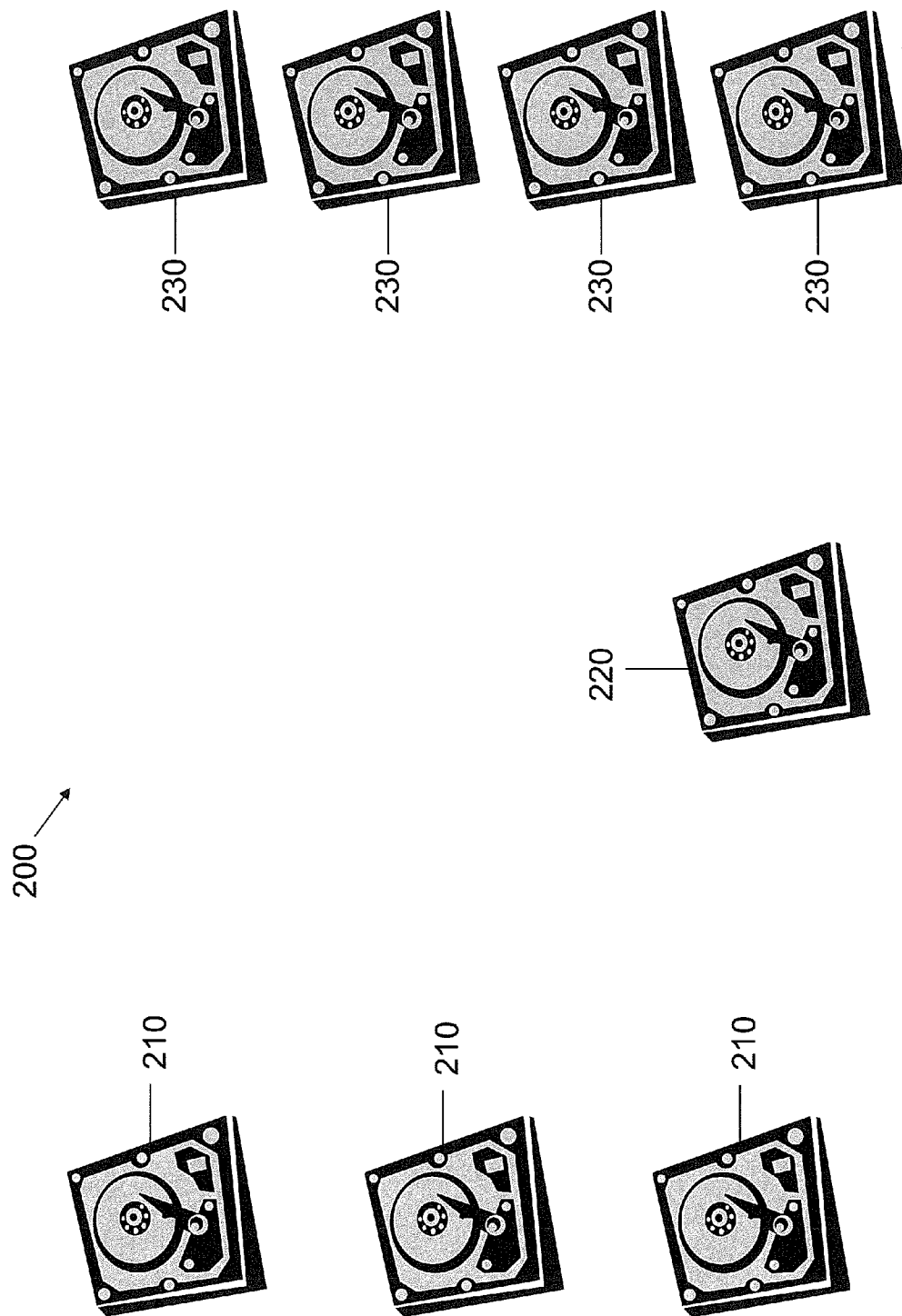


FIG. 9



US 10,003,358 B2

1

ACCELERATED ERASURE CODING SYSTEM AND METHOD

CROSS-REFERENCE TO RELATED APPLICATIONS

This application is a continuation of U.S. patent application Ser. No. 14/852,438, filed on Sep. 11, 2015, which is a continuation of U.S. patent application Ser. No. 14/223,740, filed on Mar. 24, 2014, now U.S. Pat. No. 9,160,374, issued on Oct. 13, 2015, which is a continuation of U.S. patent application Ser. No. 13/341,833, filed on Dec. 30, 2011, now U.S. Pat. No. 8,683,296, issued on Mar. 25, 2014, the entire contents of each of which are expressly incorporated herein by reference.

BACKGROUND

Field

Aspects of embodiments of the present invention are directed toward an accelerated erasure coding system and method.

Description of Related Art

An erasure code is a type of error-correcting code (ECC) useful for forward error-correction in applications like a redundant array of independent disks (RAID) or high-speed communication systems. In a typical erasure code, data (or original data) is organized in stripes, each of which is broken up into N equal-sized blocks, or data blocks, for some positive integer N . The data for each stripe is thus reconstructable by putting the N data blocks together. However, to handle situations where one or more of the original N data blocks gets lost, erasure codes also encode an additional M equal-sized blocks (called check blocks or check data) from the original N data blocks, for some positive integer M .

The N data blocks and the M check blocks are all the same size. Accordingly, there are a total of $N+M$ equal-sized blocks after encoding. The $N+M$ blocks may, for example, be transmitted to a receiver as $N+M$ separate packets, or written to $N+M$ corresponding disk drives. For ease of description, all $N+M$ blocks after encoding will be referred to as encoded blocks, though some (for example, N of them) may contain unencoded portions of the original data. That is, the encoded data refers to the original data together with the check data.

The M check blocks build redundancy into the system, in a very efficient manner, in that the original data (as well as any lost check data) can be reconstructed if any N of the $N+M$ encoded blocks are received by the receiver, or if any N of the $N+M$ disk drives are functioning correctly. Note that such an erasure code is also referred to as "optimal." For ease of description, only optimal erasure codes will be discussed in this application. In such a code, up to M of the encoded blocks can be lost, (e.g., up to M of the disk drives can fail) so that if any N of the $N+M$ encoded blocks are received successfully by the receiver, the original data (as well as the check data) can be reconstructed. $N/(N+M)$ is thus the code rate of the erasure code encoding (i.e., how much space the original data takes up in the encoded data). Erasure codes for select values of N and M can be implemented on RAID systems employing $N+M$ (disk) drives by spreading the original data among N "data" drives, and using the remaining M drives as "check" drives. Then, when any N of the $N+M$ drives are correctly functioning, the original data can be reconstructed, and the check data can be regenerated.

2

Erasure codes (or more specifically, erasure coding systems) are generally regarded as impractical for values of M larger than 1 (e.g., RAID5 systems, such as parity drive systems) or 2 (RAID6 systems), that is, for more than one or two check drives. For example, see H. Peter Anvin, "The mathematics of RAID-6," the entire content of which is incorporated herein by reference, p. 7, "Thus, in 2-disk-degraded mode, performance will be very slow. However, it is expected that that will be a rare occurrence, and that performance will not matter significantly in that case." See also Robert Maddock et al., "Surviving Two Disk Failures," p. 6, "The main difficulty with this technique is that calculating the check codes, and reconstructing data after failures, is quite complex. It involves polynomials and thus multiplication, and requires special hardware, or at least a signal processor, to do it at sufficient speed." In addition, see also James S. Plank, "All About Erasure Codes: —Reed-Solomon Coding—LDPC Coding," slide 15 (describing computational complexity of Reed-Solomon decoding), "Bottom line: When n & m grow, it is brutally expensive." Accordingly, there appears to be a general consensus among experts in the field that erasure coding systems are impractical for RAID systems for all but small values of M (that is, small numbers of check drives), such as 1 or 2.

Modern disk drives, on the other hand, are much less reliable than those envisioned when RAID was proposed. This is due to their capacity growing out of proportion to their reliability. Accordingly, systems with only a single check disk have, for the most part, been discontinued in favor of systems with two check disks.

In terms of reliability, a higher check disk count is clearly more desirable than a lower check disk count. If the count of error events on different drives is larger than the check disk count, data may be lost and that cannot be reconstructed from the correctly functioning drives. Error events extend well beyond the traditional measure of advertised mean time between failures (MTBF). A simple, real world example is a service event on a RAID system where the operator mistakenly replaces the wrong drive or, worse yet, replaces a good drive with a broken drive. In the absence of any generally accepted methodology to train, certify, and measure the effectiveness of service technicians, these types of events occur at an unknown rate, but certainly occur. The foolproof solution for protecting data in the face of multiple error events is to increase the check disk count.

SUMMARY

Aspects of embodiments of the present invention address these problems by providing a practical erasure coding system that, for byte-level RAID processing (where each byte is made up of 8 bits), performs well even for values of $N+M$ as large as 256 drives (for example, $N=127$ data drives and $M=129$ check drives). Further aspects provide for a single precomputed encoding matrix (or master encoding matrix) S of size $M_{max} \times N_{max}$, or $(N_{max} + M_{max}) \times N_{max}$ or $(M_{max} - 1) \times N_{max}$, elements (e.g., bytes), which can be used, for example, for any combination of $N \leq N_{max}$ data drives and $M \leq M_{max}$ check drives such that $N_{max} + M_{max} \leq 256$ (e.g., $N_{max}=127$ and $M_{max}=129$, or $N_{max}=63$ and $M_{max}=193$). This is an improvement over prior art solutions that rebuild such matrices from scratch every time N or M changes (such as adding another check drive). Still higher values of N and M are possible with larger processing increments, such as 2 bytes, which affords up to $N+M=65,536$ drives (such as $N=32,767$ data drives and $M=32,769$ check drives).

US 10,003,358 B2

3

Higher check disk count can offer increased reliability and decreased cost. The higher reliability comes from factors such as the ability to withstand more drive failures. The decreased cost arises from factors such as the ability to create larger groups of data drives. For example, systems with two checks disks are typically limited to group sizes of 10 or fewer drives for reliability reasons. With a higher check disk count, larger groups are available, which can lead to fewer overall components for the same unit of storage and hence, lower cost.

Additional aspects of embodiments of the present invention further address these problems by providing a standard parity drive as part of the encoding matrix. For instance, aspects provide for a parity drive for configurations with up to 127 data drives and up to 128 (non-parity) check drives, for a total of up to 256 total drives including the parity drive. Further aspects provide for different breakdowns, such as up to 63 data drives, a parity drive, and up to 192 (non-parity) check drives. Providing a parity drive offers performance comparable to RAID5 in comparable circumstances (such as single data drive failures) while also being able to tolerate significantly larger numbers of data drive failures by including additional (non-parity) check drives.

Further aspects are directed to a system and method for implementing a fast solution matrix algorithm for Reed-Solomon codes. While known solution matrix algorithms compute an $N \times N$ solution matrix (see, for example, J. S. Plank, "A tutorial on Reed-Solomon coding for fault-tolerance in RAID-like systems," *Software—Practice & Experience*, 27(9):995-1012, September 1997, and J. S. Plank and Y. Ding, "Note: Correction to the 1997 tutorial on Reed-Solomon coding," Technical Report CS-03-504, University of Tennessee, April 2003), requiring $O(N^3)$ operations, regardless of the number of failed data drives, aspects of embodiments of the present invention compute only an $F \times F$ solution matrix, where F is the number of failed data drives. The overhead for computing this $F \times F$ solution matrix is approximately $F^3/3$ multiplication operations and the same number of addition operations. Not only is $F \leq N$, in almost any practical application, the number of failed data drives F is considerably smaller than the number of data drives N . Accordingly, the fast solution matrix algorithm is considerably faster than any known approach for practical values of F and N .

Still further aspects are directed toward fast implementations of the check data generation and the lost (original and check) data reconstruction. Some of these aspects are directed toward fetching the surviving (original and check) data a minimum number of times (that is, at most once) to carry out the data reconstruction. Some of these aspects are directed toward efficient implementations that can maximize or significantly leverage the available parallel processing power of multiple cores working concurrently on the check data generation and the lost data reconstruction. Existing implementations do not attempt to accelerate these aspects of the data generation and thus fail to achieve a comparable level of performance.

In an exemplary embodiment of the present invention, a system for accelerated error-correcting code (ECC) processing is provided. The system includes a processing core for executing computer instructions and accessing data from a main memory; and a non-volatile storage medium (for example, a disk drive, or flash memory) for storing the computer instructions. The processing core, the storage medium, and the computer instructions are configured to implement an erasure coding system. The erasure coding system includes a data matrix for holding original data in the

4

main memory, a check matrix for holding check data in the main memory, an encoding matrix for holding first factors in the main memory, and a thread for executing on the processing core. The first factors are for encoding the original data into the check data. The thread includes a parallel multiplier for concurrently multiplying multiple data entries of a matrix by a single factor; and a first sequencer for ordering operations through the data matrix and the encoding matrix using the parallel multiplier to generate the check data.

The first sequencer may be configured to access each entry of the data matrix from the main memory at most once while generating the check data.

The processing core may include a plurality of processing cores. The thread may include a plurality of threads. The erasure coding system may further include a scheduler for generating the check data by dividing the data matrix into a plurality of data matrices, dividing the check matrix into a plurality of check matrices, assigning corresponding ones of the data matrices and the check matrices to the threads, and assigning the threads to the processing cores to concurrently generate portions of the check data corresponding to the check matrices from respective ones of the data matrices.

The data matrix may include a first number of rows. The check matrix may include a second number of rows. The encoding matrix may include the second number of rows and the first number of columns.

The data matrix may be configured to add rows to the first number of rows or the check matrix may be configured to add rows to the second number of rows while the first factors remain unchanged.

Each of entries of one of the rows of the encoding matrix may include a multiplicative identity factor (such as 1).

The data matrix may be configured to be divided by rows into a surviving data matrix for holding surviving original data of the original data, and a lost data matrix corresponding to lost original data of the original data and including a third number of rows. The erasure coding system may further include a solution matrix for holding second factors in the main memory. The second factors are for decoding the check data into the lost original data using the surviving original data and the first factors.

The solution matrix may include the third number of rows and the third number of columns.

The solution matrix may further include an inverted said third number by said third number sub-matrix of the encoding matrix.

The erasure coding system may further include a first list of rows of the data matrix corresponding to the surviving data matrix, and a second list of rows of the data matrix corresponding to the lost data matrix.

The data matrix may be configured to be divided into a surviving data matrix for holding surviving original data of the original data, and a lost data matrix corresponding to lost original data of the original data. The erasure coding system may further include a solution matrix for holding second factors in the main memory. The second factors are for decoding the check data into the lost original data using the surviving original data and the first factors. The thread may further include a second sequencer for ordering operations through the surviving data matrix, the encoding matrix, the check matrix, and the solution matrix using the parallel multiplier to reconstruct the lost original data.

The second sequencer may be further configured to access each entry of the surviving data matrix from the main memory at most once while reconstructing the lost original data.

US 10,003,358 B2

5

The processing core may include a plurality of processing cores. The thread may include a plurality of threads. The erasure coding system may further include: a scheduler for generating the check data and reconstructing the lost original data by dividing the data matrix into a plurality of data matrices; dividing the surviving data matrix into a plurality of surviving data matrices; dividing the lost data matrix into a plurality of lost data matrices; dividing the check matrix into a plurality of check matrices; assigning corresponding ones of the data matrices, the surviving data matrices, the lost data matrices, and the check matrices to the threads; and assigning the threads to the processing cores to concurrently generate portions of the check data corresponding to the check matrices from respective ones of the data matrices and to concurrently reconstruct portions of the lost original data corresponding to the lost data matrices from respective ones of the surviving data matrices and the check matrices.

The check matrix may be configured to be divided into a surviving check matrix for holding surviving check data of the check data, and a lost check matrix corresponding to lost check data of the check data. The second sequencer may be configured to order operations through the surviving data matrix, the reconstructed lost original data, and the encoding matrix using the parallel multiplier to regenerate the lost check data.

The second sequencer may be further configured to reconstruct the lost original data concurrently with regenerating the lost check data.

The second sequencer may be further configured to access each entry of the surviving data matrix from the main memory at most once while reconstructing the lost original data and regenerating the lost check data.

The second sequencer may be further configured to regenerate the lost check data without accessing the reconstructed lost original data from the main memory.

The processing core may include a plurality of processing cores. The thread may include a plurality of threads. The erasure coding system may further include a scheduler for generating the check data, reconstructing the lost original data, and regenerating the lost check data by: dividing the data matrix into a plurality of data matrices; dividing the surviving data matrix into a plurality of surviving data matrices; dividing the lost data matrix into a plurality of lost data matrices; dividing the check matrix into a plurality of check matrices; dividing the surviving check matrix into a plurality of surviving check matrices; dividing the lost check matrix into a plurality of lost check matrices; assigning corresponding ones of the data matrices, the surviving data matrices, the lost data matrices, the check matrices, the surviving check matrices, and the lost check matrices to the threads; and assigning the threads to the processing cores to concurrently generate portions of the check data corresponding to the check matrices from respective ones of the data matrices, to concurrently reconstruct portions of the lost original data corresponding to the lost data matrices from respective ones of the surviving data matrices and the surviving check matrices, and to concurrently regenerate portions of the lost check data corresponding to the lost check matrices from respective ones of the surviving data matrices and respective portions of the reconstructed lost original data.

The processing core may include 16 data registers. Each of the data registers may include 16 bytes. The parallel multiplier may be configured to process the data in units of at least 64 bytes spread over at least four of the data registers at a time.

6

Consecutive instructions to process each of the units of the data may access separate ones of the data registers to permit concurrent execution of the consecutive instructions by the processing core.

The parallel multiplier may include two lookup tables for doing concurrent multiplication of 4-bit quantities across 16 byte-sized entries using the PSHUFB (Packed Shuffle Bytes) instruction.

The parallel multiplier may be further configured to receive an input operand in four of the data registers, and return with the input operand intact in the four of the data registers.

According to another exemplary embodiment of the present invention, a method of accelerated error-correcting code (ECC) processing on a computing system is provided. The computing system includes a non-volatile storage medium (such as a disk drive or flash memory), a processing core for accessing instructions and data from a main memory, and a computer program including a plurality of computer instructions for implementing an erasure coding system. The method includes: storing the computer program on the storage medium; executing the computer instructions on the processing core; arranging original data as a data matrix in the main memory; arranging first factors as an encoding matrix in the main memory, the first factors being for encoding the original data into check data, the check data being arranged as a check matrix in the main memory; and generating the check data using a parallel multiplier for concurrently multiplying multiple data entries of a matrix by a single factor. The generating of the check data includes ordering operations through the data matrix and the encoding matrix using the parallel multiplier.

The generating of the check data may include accessing each entry of the data matrix from the main memory at most once.

The processing core may include a plurality of processing cores. The executing of the computer instructions may include executing the computer instructions on the processing cores. The method may further include scheduling the generating of the check data by: dividing the data matrix into a plurality of data matrices; dividing the check matrix into a plurality of check matrices; and assigning corresponding ones of the data matrices and the check matrices to the processing cores to concurrently generate portions of the check data corresponding to the check matrices from respective ones of the data matrices.

The method may further include: dividing the data matrix into a surviving data matrix for holding surviving original data of the original data, and a lost data matrix corresponding to lost original data of the original data; arranging second factors as a solution matrix in the main memory, the second factors being for decoding the check data into the lost original data using the surviving original data and the first factors; and reconstructing the lost original data by ordering operations through the surviving data matrix, the encoding matrix, the check matrix, and the solution matrix using the parallel multiplier.

The reconstructing of the lost original data may include accessing each entry of the surviving data matrix from the main memory at most once.

The processing core may include a plurality of processing cores. The executing of the computer instructions may include executing the computer instructions on the processing cores. The method may further include scheduling the generating of the check data and the reconstructing of the lost original data by: dividing the data matrix into a plurality of data matrices; dividing the surviving data matrix into a

US 10,003,358 B2

7

plurality of surviving data matrices; dividing the lost data matrix into a plurality of lost data matrices; dividing the check matrix into a plurality of check matrices; and assigning corresponding ones of the data matrices, the surviving data matrices, the lost data matrices, and the check matrices to the processing cores to concurrently generate portions of the check data corresponding to the check matrices from respective ones of the data matrices and to concurrently reconstruct portions of the lost original data corresponding to the lost data matrices from respective ones of the surviving data matrices and the check matrices.

The method may further include: dividing the check matrix into a surviving check matrix for holding surviving check data of the check data, and a lost check matrix corresponding to lost check data of the check data; and regenerating the lost check data by ordering operations through the surviving data matrix, the reconstructed lost original data, and the encoding matrix using the parallel multiplier.

The reconstructing of the lost original data may take place concurrently with the regenerating of the lost check data.

The reconstructing of the lost original data and the regenerating of the lost check data may include accessing each entry of the surviving data matrix from the main memory at most once.

The regenerating of the lost check data may take place without accessing the reconstructed lost original data from the main memory.

The processing core may include a plurality of processing cores. The executing of the computer instructions may include executing the computer instructions on the processing cores. The method may further include scheduling the generating of the check data, the reconstructing of the lost original data, and the regenerating of the lost check data by: dividing the data matrix into a plurality of data matrices; dividing the surviving data matrix into a plurality of surviving data matrices; dividing the lost data matrix into a plurality of lost data matrices; dividing the check matrix into a plurality of check matrices; dividing the surviving check matrix into a plurality of surviving check matrices; dividing the lost check matrix into a plurality of lost check matrices; and assigning corresponding ones of the data matrices, the surviving data matrices, the lost data matrices, the check matrices, the surviving check matrices, and the lost check matrices to the processing cores to concurrently generate portions of the check data corresponding to the check matrices from respective ones of the data matrices, to concurrently reconstruct portions of the lost original data corresponding to the lost data matrices from respective ones of the surviving data matrices and the surviving check matrices, and to concurrently regenerate portions of the lost check data corresponding to the lost check matrices from respective ones of the surviving data matrices and respective portions of the reconstructed lost original data.

According to yet another exemplary embodiment of the present invention, a non-transitory computer-readable storage medium (such as a disk drive, a compact disk (CD), a digital video disk (DVD), flash memory, a universal serial bus (USB) drive, etc.) containing a computer program including a plurality of computer instructions for performing accelerated error-correcting code (ECC) processing on a computing system is provided. The computing system includes a processing core for accessing instructions and data from a main memory. The computer instructions are configured to implement an erasure coding system when executed on the computing system by performing the steps of: arranging original data as a data matrix in the main

8

memory; arranging first factors as an encoding matrix in the main memory, the first factors being for encoding the original data into check data, the check data being arranged as a check matrix in the main memory; and generating the check data using a parallel multiplier for concurrently multiplying multiple data entries of a matrix by a single factor. The generating of the check data includes ordering operations through the data matrix and the encoding matrix using the parallel multiplier.

The generating of the check data may include accessing each entry of the data matrix from the main memory at most once.

The processing core may include a plurality of processing cores. The computer instructions may be further configured to perform the step of scheduling the generating of the check data by: dividing the data matrix into a plurality of data matrices; dividing the check matrix into a plurality of check matrices; and assigning corresponding ones of the data matrices and the check matrices to the processing cores to concurrently generate portions of the check data corresponding to the check matrices from respective ones of the data matrices.

The computer instructions may be further configured to perform the steps of: dividing the data matrix into a surviving data matrix for holding surviving original data of the original data, and a lost data matrix corresponding to lost original data of the original data; arranging second factors as a solution matrix in the main memory, the second factors being for decoding the check data into the lost original data using the surviving original data and the first factors; and reconstructing the lost original data by ordering operations through the surviving data matrix, the encoding matrix, the check matrix, and the solution matrix using the parallel multiplier.

The computer instructions may be further configured to perform the steps of: dividing the check matrix into a surviving check matrix for holding surviving check data of the check data, and a lost check matrix corresponding to lost check data of the check data; and regenerating the lost check data by ordering operations through the surviving data matrix, the reconstructed lost original data, and the encoding matrix using the parallel multiplier.

The reconstructing of the lost original data and the regenerating of the lost check data may include accessing each entry of the surviving data matrix from the main memory at most once.

The processing core may include a plurality of processing cores. The computer instructions may be further configured to perform the step of scheduling the generating of the check data, the reconstructing of the lost original data, and the regenerating of the lost check data by: dividing the data matrix into a plurality of data matrices; dividing the surviving data matrix into a plurality of surviving data matrices; dividing the lost data matrix into a plurality of lost data matrices; dividing the check matrix into a plurality of check matrices; dividing the surviving check matrix into a plurality of surviving check matrices; dividing the lost check matrix into a plurality of lost check matrices; and assigning corresponding ones of the data matrices, the surviving data matrices, the lost data matrices, the check matrices, the surviving check matrices, and the lost check matrices to the processing cores to concurrently generate portions of the check data corresponding to the check matrices from respective ones of the data matrices, to concurrently reconstruct portions of the lost original data corresponding to the lost data matrices from respective ones of the surviving data matrices and the surviving check matrices, and to concur-

rently regenerate portions of the lost check data corresponding to the lost check matrices from respective ones of the surviving data matrices and respective portions of the reconstructed lost original data.

By providing practical and efficient systems and methods for erasure coding systems (which for byte-level processing can support up to $N+M=256$ drives, such as $N=127$ data drives and $M=129$ check drives, including a parity drive), applications such as RAID systems that can tolerate far more failing drives than was thought to be possible or practical can be implemented with accelerated performance significantly better than any prior art solution.

BRIEF DESCRIPTION OF THE DRAWINGS

The accompanying drawings, together with the specification, illustrate exemplary embodiments of the present invention and, together with the description, serve to explain aspects and principles of the present invention.

FIG. 1 shows an exemplary stripe of original and check data according to an embodiment of the present invention.

FIG. 2 shows an exemplary method for reconstructing lost data after a failure of one or more drives according to an embodiment of the present invention.

FIG. 3 shows an exemplary method for performing a parallel lookup Galois field multiplication according to an embodiment of the present invention.

FIG. 4 shows an exemplary method for sequencing the parallel lookup multiplier to perform the check data generation according to an embodiment of the present invention.

FIGS. 5-7 show an exemplary method for sequencing the parallel lookup multiplier to perform the lost data reconstruction according to an embodiment of the present invention.

FIG. 8 illustrates a multi-core architecture system according to an embodiment of the present invention.

FIG. 9 shows an exemplary disk drive configuration according to an embodiment of the present invention.

DETAILED DESCRIPTION

Hereinafter, exemplary embodiments of the invention will be described in more detail with reference to the accompanying drawings. In the drawings, like reference numerals refer to like elements throughout.

While optimal erasure codes have many applications, for ease of description, they will be described in this application with respect to RAID applications, i.e., erasure coding systems for the storage and retrieval of digital data distributed across numerous storage devices (or drives), though the present application is not limited thereto. For further ease of description, the storage devices will be assumed to be disk drives, though the invention is not limited thereto. In RAID systems, the data (or original data) is broken up into stripes, each of which includes N uniformly sized blocks (data blocks), and the N blocks are written across N separate drives (the data drives), one block per data drive.

In addition, for ease of description, blocks will be assumed to be composed of L elements, each element having a fixed size, say 8 bits or one byte. An element, such as a byte, forms the fundamental unit of operation for the RAID processing, but the invention is just as applicable to other size elements, such as 16 bits (2 bytes). For simplification, unless otherwise indicated, elements will be assumed to be one byte in size throughout the description that follows, and the term “element(s)” and “byte(s)” will be used synonymously.

Conceptually, different stripes can distribute their data blocks across different combinations of drives, or have different block sizes or numbers of blocks, etc., but for

simplification and ease of description and implementation, the described embodiments in the present application assume a consistent block size (L bytes) and distribution of blocks among the data drives between stripes. Further, all variables, such as the number of data drives N , will be assumed to be positive integers unless otherwise specified. In addition, since the $N=1$ case reduces to simple data mirroring (that is, copying the same data drive multiple times), it will also be assumed for simplicity that $N \geq 2$ throughout.

The N data blocks from each stripe are combined using arithmetic operations (to be described in more detail below) in M different ways to produce M blocks of check data (check blocks), and the M check blocks written across M drives (the check drives) separate from the N data drives, one block per check drive. These combinations can take place, for example, when new (or changed) data is written to (or back to) disk. Accordingly, each of the $N+M$ drives (data drives and check drives) stores a similar amount of data, namely one block for each stripe. As the processing of multiple stripes is conceptually similar to the processing of one stripe (only processing multiple blocks per drive instead of one), it will be further assumed for simplification that the data being stored or retrieved is only one stripe in size unless otherwise indicated. It will also be assumed that the block size L is sufficiently large that the data can be consistently divided across each block to produce subsets of the data that include respective portions of the blocks (for efficient concurrent processing by different processing units).

FIG. 1 shows an exemplary stripe **10** of original and check data according to an embodiment of the present invention.

Referring to FIG. 1, the stripe **10** can be thought of not only as the original N data blocks **20** that make up the original data, but also the corresponding M check blocks **30** generated from the original data (that is, the stripe **10** represents encoded data). Each of the N data blocks **20** is composed of L bytes **25** (labeled byte 1, byte 2, . . . , byte L), and each of the M check blocks **30** is composed of L bytes **35** (labeled similarly). In addition, check drive 1, byte 1, is a linear combination of data drive 1, byte 1; data drive 2, byte 1; . . . ; data drive N , byte 1. Likewise, check drive 1, byte 2, is generated from the same linear combination formula as check drive 1, byte 1, only using data drive 1, byte 2; data drive 2, byte 2; . . . ; data drive N , byte 2. In contrast, check drive 2, byte 1, uses a different linear combination formula than check drive 1, byte 1, but applies it to the same data, namely data drive 1, byte 1; data drive 2, byte 1; . . . ; data drive N , byte 1. In this fashion, each of the other check bytes **35** is a linear combination of the respective bytes of each of the N data drives **20** and using the corresponding linear combination formula for the particular check drive **30**.

The stripe **10** in FIG. 1 can also be represented as a matrix C of encoded data. C has two sub-matrices, namely original data D on top and check data J on bottom. That is,

$$C = \begin{bmatrix} D \\ J \end{bmatrix} = \begin{bmatrix} D_{11} & D_{12} & \dots & D_{1L} \\ D_{21} & D_{22} & \dots & D_{2L} \\ \vdots & \vdots & \ddots & \vdots \\ D_{N1} & D_{N2} & \dots & D_{NL} \\ J_{11} & J_{12} & \dots & J_{1L} \\ J_{21} & J_{22} & \dots & J_{2L} \\ \vdots & \vdots & \ddots & \vdots \\ J_{M1} & J_{M2} & \dots & J_{ML} \end{bmatrix},$$

where D_{ij} =byte j from data drive i and J_{ij} =byte j from check drive i . Thus, the rows of encoded data C represent blocks, while the columns represent corresponding bytes of each of the drives.

US 10,003,358 B2

11

Further, in case of a disk drive failure of one or more disks, the arithmetic operations are designed in such a fashion that for any stripe, the original data (and by extension, the check data) can be reconstructed from any combination of N data and check blocks from the corresponding N+M data and check blocks that comprise the stripe. Thus, RAID provides both parallel processing (reading and writing the data in stripes across multiple drives concurrently) and fault tolerance (regeneration of the original data even if as many as M of the drives fail), at the computational cost of generating the check data any time new data is written to disk, or changed data is written back to disk, as well as the computational cost of reconstructing any lost original data and regenerating any lost check data after a disk failure.

For example, for M=1 check drive, a single parity drive can function as the check drive (i.e., a RAID4 system). Here, the arithmetic operation is bitwise exclusive OR of each of the N corresponding data bytes in each data block of the stripe. In addition, as mentioned earlier, the assignment of parity blocks from different stripes to the same drive (i.e., RAID4) or different drives (i.e., RAID5) is arbitrary, but it does simplify the description and implementation to use a consistent assignment between stripes, so that will be assumed throughout. Since M=1 reduces to the case of a single parity drive, it will further be assumed for simplicity that M≥2 throughout.

For such larger values of M, Galois field arithmetic is used to manipulate the data, as described in more detail later. Galois field arithmetic, for Galois fields of powers-of-2 (such as 2^p) numbers of elements, includes two fundamental operations: (1) addition (which is just bitwise exclusive OR, as with the parity drive-only operations for M=1), and (2) multiplication. While Galois field (GF) addition is trivial on standard processors, GF multiplication is not. Accordingly, a significant component of RAID performance for M≥2 is speeding up the performance of GF multiplication, as will be discussed later. For purposes of description, GF addition will be represented by the symbol+throughout while GF multiplication will be represented by the symbol×throughout.

Briefly, in exemplary embodiments of the present invention, each of the M check drives holds linear combinations (over GF arithmetic) of the N data drives of original data, one linear combination (i.e., a GF sum of N terms, where each term represents a byte of original data times a corresponding factor (using GF multiplication) for the respective data drive) for each check drive, as applied to respective bytes in each block. One such linear combination can be a simple parity, i.e., entirely GF addition (all factors equal 1), such as a GF sum of the first byte in each block of original data as described above.

The remaining M-1 linear combinations include more involved calculations that include the nontrivial GF multiplication operations (e.g., performing a GF multiplication of the first byte in each block by a corresponding factor for the respective data drive, and then performing a GF sum of all these products). These linear combinations can be represented by an (N+M)×N matrix (encoding matrix or information dispersal matrix (IDM)) E of the different factors, one factor for each combination of (data or check) drive and data drive, with one row for each of the N+M data and check drives and one column for each of the N data drives. The IDM E can also be represented as

$$\begin{bmatrix} I_N \\ H \end{bmatrix},$$

where I_N represents the N×N identity matrix (i.e., the original (unencoded) data) and H represents the M×N matrix of

12

factors for the check drives (where each of the M rows corresponds to one of the M check drives and each of the N columns corresponds to one of the N data drives).

Thus,

$$E = \begin{bmatrix} I_N \\ H \end{bmatrix} = \begin{bmatrix} 1 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 1 \\ H_{11} & H_{12} & \dots & H_{1N} \\ H_{21} & H_{22} & \dots & H_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ H_{M1} & H_{M2} & \dots & H_{MN} \end{bmatrix},$$

where H_{ij}=factor for check drive i and data drive j. Thus, the rows of encoded data C represent blocks, while the columns represent corresponding bytes of each of the drives. In addition, check factors H, original data D, and check data J are related by the formula J=H×D (that is, matrix multiplication), or

$$\begin{bmatrix} J_{11} & J_{12} & \dots & J_{1L} \\ J_{21} & J_{22} & \dots & J_{2L} \\ \vdots & \vdots & \ddots & \vdots \\ J_{M1} & J_{M2} & \dots & J_{ML} \end{bmatrix} = \begin{bmatrix} H_{11} & H_{12} & \dots & H_{1N} \\ H_{21} & H_{22} & \dots & H_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ H_{M1} & H_{M2} & \dots & H_{MN} \end{bmatrix} \times \begin{bmatrix} D_{11} & D_{12} & \dots & D_{1L} \\ D_{21} & D_{22} & \dots & D_{2L} \\ \vdots & \vdots & \ddots & \vdots \\ D_{N1} & D_{N2} & \dots & D_{NL} \end{bmatrix}$$

where J₁₁=(H₁₁×D₁₁)+(H₁₂×D₂₁)+ . . . +(H_{1N}×D_{N1}), J₁₂=(H₁₁×D₁₂)+(H₁₂×D₂₂)+ . . . +(H_{1N}×D_{N2}), J₂₁=(H₂₁×D₁₁)+(H₂₂×D₂₁)+ . . . +(H_{2N}×D_{N1}), and in general, J_{ij}=(H₁₁×D_{1j})+(H₁₂×D_{2j})+ . . . +(H_{iN}×D_{Nj}) for 1≤i≤M and 1≤j≤L.

Such an encoding matrix E is also referred to as an information dispersal matrix (IDM). It should be noted that matrices such as check drive encoding matrix H and identity matrix I_N also represent encoding matrices, in that they represent matrices of factors to produce linear combinations over GF arithmetic of the original data. In practice, the identity matrix I_N is trivial and may not need to be constructed as part of the IDM E. Only the encoding matrix E, however, will be referred to as the IDM. Methods of building an encoding matrix such as IDM E or check drive encoding matrix H are discussed below. In further embodiments of the present invention (as discussed further in Appendix A), such (N+M)×N (or M×N) matrices can be trivially constructed (or simply indexed) from a master encoding matrix S, which is composed of (N_{max}+M_{max})×N_{max} (or M_{max}×N_{max}) bytes or elements, where N_{max}+M_{max}=256 (or some other power of two) and N≤N_{max} and M≤M_{max}. For example, one such master encoding matrix S can include a 127×127 element identity matrix on top (for up to N_{max}=127 data drives), a row of 1's (for a parity drive), and a 128×127 element encoding matrix on bottom (for up to M_{max}=129 check drives, including the parity drive), for a total of N_{max}+M_{max}=256 drives.

The original data, in turn, can be represented by an N×L matrix D of bytes, each of the N rows representing the L bytes of a block of the corresponding one of the N data drives. If C represents the corresponding (N+M)×L matrix

US 10,003,358 B2

13

of encoded bytes (where each of the N+M rows corresponds to one of the N+M data and check drives), then C can be represented as E×

$$D = \begin{bmatrix} I_N \\ H \end{bmatrix} \times D = \begin{bmatrix} I_N \times D \\ H \times D \end{bmatrix} = \begin{bmatrix} D \\ J \end{bmatrix},$$

where J=H×D is an M×L matrix of check data, with each of the M rows representing the L check bytes of the corresponding one of the M check drives. It should be noted that in the relationships such as C=E×D or J=H×D, × represents matrix multiplication over the Galois field (i.e., GF multiplication and GF addition being used to generate each of the entries in, for example, C or J).

In exemplary embodiments of the present invention, the first row of the check drive encoding matrix H (or the (N+1)th row of the IDM E) can be all 1's, representing the parity drive. For linear combinations involving this row, the GF multiplication can be bypassed and replaced with a GF sum of the corresponding bytes since the products are all trivial products involving the identity element 1. Accordingly, in parity drive implementations, the check drive encoding matrix H can also be thought of as an (M-1)×N matrix of non-trivial factors (that is, factors intended to be used in GF multiplication and not just GF addition).

Much of the RAID processing involves generating the check data when new or changed data is written to (or back to) disk. The other significant event for RAID processing is when one or more of the drives fail (data or check drives), or for whatever reason become unavailable. Assume that in such a failure scenario, F data drives fail and G check drives fail, where F and G are nonnegative integers. If F=0, then only check drives failed and all of the original data D survived. In this case, the lost check data can be regenerated from the original data D.

Accordingly, assume at least one data drive fails, that is, F≥1, and let K=N-F represent the number of data drives that survive. K is also a nonnegative integer. In addition, let X represent the surviving original data and Y represent the lost original data. That is, X is a K×L matrix composed of the K rows of the original data matrix D corresponding to the K surviving data drives, while Y is an F×L matrix composed of the F rows of the original data matrix D corresponding to the F failed data drives.

$$\begin{bmatrix} X \\ Y \end{bmatrix}$$

thus represents a permuted original data matrix D' (that is, the original data matrix D, only with the surviving original data X on top and the lost original data Y on bottom. It should be noted that once the lost original data Y is reconstructed, it can be combined with the surviving original data X to restore the original data D, from which the check data for any of the failed check drives can be regenerated.

It should also be noted that M-G check drives survive. In order to reconstruct the lost original data Y, enough (that is, at least N) total drives must survive. Given that K=N-F data drives survive, and that M-G check drives survive, it follows that (N-F)+(M-G)≥N must be true to reconstruct the lost original data Y. This is equivalent to F+G≤M (i.e., no more than F+G drives fail), or F≤M-G (that is, the number of failed data drives does not exceed the number of surviving check drives). It will therefore be assumed for simplicity that F≤M-G.

In the routines that follow, performance can be enhanced by prebuilding lists of the failed and surviving data and check drives (that is, four separate lists). This allows pro-

14

cessing of the different sets of surviving and failed drives to be done more efficiently than existing solutions, which use, for example, bit vectors that have to be examined one bit at a time and often include large numbers of consecutive zeros (or ones) when ones (or zeros) are the bit values of interest.

FIG. 2 shows an exemplary method 300 for reconstructing lost data after a failure of one or more drives according to an embodiment of the present invention.

While the recovery process is described in more detail later, briefly it consists of two parts: (1) determining the solution matrix, and (2) reconstructing the lost data from the surviving data. Determining the solution matrix can be done in three steps with the following algorithm (Algorithm 1), with reference to FIG. 2:

1. (Step 310 in FIG. 2) Reducing the (M+N)×N IDM E to an N×N reduced encoding matrix T (also referred to as the transformed IDM) including the K surviving data drive rows and any F of the M-G surviving check drive rows (for instance, the first F surviving check drive rows, as these will include the parity drive if it survived; recall that F≤M-G was assumed). In addition, the columns of the reduced encoding matrix T are rearranged so that the K columns corresponding to the K surviving data drives are on the left side of the matrix and the F columns corresponding to the F failed drives are on the right side of the matrix. (Step 320) These F surviving check drives selected to rebuild the lost original data Y will henceforth be referred to as "the F surviving check drives," and their check data W will be referred to as "the surviving check data," even though M-G check drives survived. It should be noted that W is an F×L matrix composed of the F rows of the check data J corresponding to the F surviving check drives. Further, the surviving encoded data can be represented as a sub-matrix C' of the encoded data C. The surviving encoded data C' is an N×L matrix composed of the surviving original data X on top and the surviving check data W on bottom, that is,

$$C' = \begin{bmatrix} X \\ W \end{bmatrix}.$$

2. (Step 330) Splitting the reduced encoding matrix T into four sub-matrices (that are also encoding matrices): (i) a K×K identity matrix I_K (corresponding to the K surviving data drives) in the upper left, (ii) a K×F matrix O of zeros in the upper right, (iii) an F×K encoding matrix A in the lower left corresponding to the F surviving check drive rows and the K surviving data drive columns, and (iv) an F×F encoding matrix B in the lower right corresponding to the F surviving check drive rows and the F failed data drive columns. Thus, the reduced encoding matrix T can be represented as

$$\begin{bmatrix} I_K & O \\ A & B \end{bmatrix}.$$

3. (Step 340) Calculating the inverse B⁻¹ of the F×F encoding matrix B. As is shown in more detail in Appendix A, C'=T×D', or

$$\begin{bmatrix} X \\ W \end{bmatrix} = \begin{bmatrix} I_K & O \\ A & B \end{bmatrix} \times \begin{bmatrix} X \\ Y \end{bmatrix},$$

which is mathematically equivalent to W=A×X+B×Y. B⁻¹ is the solution matrix, and is itself an F×F encoding matrix. Calculating the solution matrix B⁻¹ thus allows the lost

original data Y to be reconstructed from the encoding matrices A and B along with the surviving original data X and the surviving check data W.

The $F \times K$ encoding matrix A represents the original encoding matrix E, only limited to the K surviving data drives and the F surviving check drives. That is, each of the F rows of A represents a different one of the F surviving check drives, while each of the K columns of A represents a different one of the K surviving data drives. Thus, A provides the encoding factors needed to encode the original data for the surviving check drives, but only applied to the surviving data drives (that is, the surviving partial check data). Since the surviving original data X is available, A can be used to generate this surviving partial check data.

In similar fashion, the $F \times F$ encoding matrix B represents the original encoding matrix E, only limited to the F surviving check drives and the F failed data drives. That is, the F rows of B correspond to the same F rows of A, while each of the F columns of B represents a different one of the F failed data drives. Thus, B provides the encoding factors needed to encode the original data for the surviving check drives, but only applied to the failed data drives (that is, the lost partial check data). Since the lost original data Y is not available, B cannot be used to generate any of the lost partial check data. However, this lost partial check data can be determined from A and the surviving check data W. Since this lost partial check data represents the result of applying B to the lost original data Y, B^{-1} thus represents the necessary factors to reconstruct the lost original data Y from the lost partial check data.

It should be noted that steps 1 and 2 in Algorithm 1 above are logical, in that encoding matrices A and B (or the reduced encoding matrix T, for that matter) do not have to actually be constructed. Appropriate indexing of the IDM E (or the master encoding matrix 5) can be used to obtain any of their entries. Step 3, however, is a matrix inversion over GF arithmetic and takes $O(F^3)$ operations, as discussed in more detail later. Nonetheless, this is a significant improvement over existing solutions, which require $O(N^3)$ operations, since the number of failed data drives F is usually significantly less than the number of data drives N in any practical situation.

(Step 350 in FIG. 2) Once the encoding matrix A and the solution matrix B^{-1} are known, reconstructing the lost data from the surviving data (that is, the surviving original data X and the surviving check data W) can be accomplished in four steps using the following algorithm (Algorithm 2):

1. Use A and the surviving original data X (using matrix multiplication) to generate the surviving check data (i.e., $A \times X$), only limited to the K surviving data drives. Call this limited check data the surviving partial check data.
2. Subtract this surviving partial check data from the surviving check data W (using matrix subtraction, i.e., $W - A \times X$, which is just entry-by-entry GF subtraction, which is the same as GF addition for this Galois field). This generates the surviving check data, only this time limited to the F failed data drives. Call this limited check data the lost partial check data.
3. Use the solution matrix B^{-1} and the lost partial check data (using matrix multiplication, i.e., $B^{-1} \times (W - A \times X)$) to reconstruct the lost original data Y. Call this the recovered original data Y.
4. Use the corresponding rows of the IDM E (or master encoding matrix S) for each of the G failed check drives along with the original data D, as reconstructed from

the surviving and recovered original data X and Y, to regenerate the lost check data (using matrix multiplication).

As will be shown in more detail later, steps 1-3 together require $O(F)$ operations times the amount of original data D to reconstruct the lost original data Y for the F failed data drives (i.e., roughly 1 operation per failed data drive per byte of original data D), which is proportionally equivalent to the $O(M)$ operations times the amount of original data D needed to generate the check data J for the M check drives (i.e., roughly 1 operation per check drive per byte of original data D). In addition, this same equivalence extends to step 4, which takes $O(G)$ operations times the amount of original data D needed to regenerate the lost check data for the G failed check drives (i.e., roughly 1 operation per failed check drive per byte of original data D). In summary, the number of operations needed to reconstruct the lost data is $O(F+G)$ times the amount of original data D (i.e., roughly 1 operation per failed drive (data or check) per byte of original data D). Since $F+G \leq M$, this means that the computational complexity of Algorithm 2 (reconstructing the lost data from the surviving data) is no more than that of generating the check data J from the original data D.

As mentioned above, for exemplary purposes and ease of description, data is assumed to be organized in 8-bit bytes, each byte capable of taking on $2^8=256$ possible values. Such data can be manipulated in byte-size elements using GF arithmetic for a Galois field of size $2^8=256$ elements. It should also be noted that the same mathematical principles apply to any power-of-two 2^P number of elements, not just 256, as Galois fields can be constructed for any integral power of a prime number. Since Galois fields are finite, and since GF operations never overflow, all results are the same size as the inputs, for example, 8 bits.

In a Galois field of a power-of-two number of elements, addition and subtraction are the same operation, namely a bitwise exclusive OR (XOR) of the two operands. This is a very fast operation to perform on any current processor. It can also be performed on multiple bytes concurrently. Since the addition and subtraction operations take place, for example, on a byte-level basis, they can be done in parallel by using, for instance, x86 architecture Streaming SIMD Extensions (SSE) instructions (SIMD stands for single instruction, multiple data, and refers to performing the same instruction on different pieces of data, possibly concurrently), such as PXOR (Packed (bitwise) Exclusive OR).

SSE instructions can process, for example, 16-byte registers (XMM registers), and are able to process such registers as though they contain 16 separate one-byte operands (or 8 separate two-byte operands, or four separate four-byte operands, etc.) Accordingly, SSE instructions can do byte-level processing 16 times faster than when compared to processing a byte at a time. Further, there are 16 XMM registers, so dedicating four such registers for operand storage allows the data to be processed in 64-byte increments, using the other 12 registers for temporary storage. That is, individual operations can be performed as four consecutive SSE operations on the four respective registers (64 bytes), which can often allow such instructions to be efficiently pipelined and/or concurrently executed by the processor. In addition, the SSE instructions allows the same processing to be performed on different such 64-byte increments of data in parallel using different cores. Thus, using four separate cores can potentially speed up this processing by an additional factor of 4 over using a single core.

For example, a parallel adder (Parallel Adder) can be built using the 16-byte XMM registers and four consecutive

US 10,003,358 B2

17

PXOR instructions. Such parallel processing (that is, 64 bytes at a time with only a few machine-level instructions) for GF arithmetic is a significant improvement over doing the addition one byte at a time. Since the data is organized in blocks of any fixed number of bytes, such as 4096 bytes (4 kilobytes, or 4 KB) or 32,768 bytes (32 KB), a block can be composed of numerous such 64-byte chunks (e.g., 64 separate 64-byte chunks in 4 KB, or 512 chunks in 32 KB).

Multiplication in a Galois field is not as straightforward. While much of it is bitwise shifts and exclusive OR's (i.e., "additions") that are very fast operations, the numbers "wrap" in peculiar ways when they are shifted outside of their normal bounds (because the field has only a finite set of elements), which can slow down the calculations. This "wrapping" in the GF multiplication can be addressed in many ways. For example, the multiplication can be implemented serially (Serial Multiplier) as a loop iterating over the bits of one operand while performing the shifts, adds, and wraps on the other operand. Such processing, however, takes several machine instructions per bit for 8 separate bits. In other words, this technique requires dozens of machine instructions per byte being multiplied. This is inefficient compared to, for example, the performance of the Parallel Adder described above.

For another approach (Serial Lookup Multiplier), multiplication tables (of all the possible products, or at least all the non-trivial products) can be pre-computed and built ahead of time. For example, a table of $256 \times 256 = 65,536$ bytes can hold all the possible products of the two different one-byte operands). However, such tables can force serialized access on what are only byte-level operations, and not take advantage of wide (concurrent) data paths available on modern processors, such as those used to implement the Parallel Adder above.

In still another approach (Parallel Multiplier), the GF multiplication can be done on multiple bytes at a time, since the same factor in the encoding matrix is multiplied with every element in a data block. Thus, the same factor can be multiplied with 64 consecutive data block bytes at a time. This is similar to the Parallel Adder described above, only there are several more operations needed to perform the operation. While this can be implemented as a loop on each bit of the factor, as described above, only performing the shifts, adds, and wraps on 64 bytes at a time, it can be more efficient to process the 256 possible factors as a (C language) switch statement, with inline code for each of 256 different combinations of two primitive GF operations: Multiply-by-2 and Add. For example, GF multiplication by the factor 3 can be effected by first doing a Multiply-by-2 followed by an Add. Likewise, GF multiplication by 4 is just a Multiply-by-2 followed by a Multiply-by-2 while multiplication by 6 is a Multiply-by-2 followed by an Add and then by another Multiply-by-2.

While this Add is identical to the Parallel Adder described above (e.g., four consecutive PXOR instructions to process 64 separate bytes), Multiply-by-2 is not as straightforward. For example, Multiply-by-2 in GF arithmetic can be implemented across 64 bytes at a time in 4 XMM registers via 4 consecutive PXOR instructions, 4 consecutive PCMPGTB (Packed Compare for Greater Than) instructions, 4 consecutive PADDB (Packed Add) instructions, 4 consecutive PAND (Bitwise AND) instructions, and 4 consecutive PXOR instructions. Though this takes 20 machine instructions, the instructions are very fast and results in 64 consecutive bytes of data at a time being multiplied by 2.

For 64 bytes of data, assuming a random factor between 0 and 255, the total overhead for the Parallel Multiplier is

18

about 6 calls to multiply-by-2 and about 3.5 calls to add, or about $6 \times 20 + 3.5 \times 4 = 134$ machine instructions, or a little over 2 machine instructions per byte of data. While this compares favorably with byte-level processing, it is still possible to improve on this by building a parallel multiplier with a table lookup (Parallel Lookup Multiplier) using the PSHUFB (Packed Shuffle Bytes) instruction and doing the GF multiplication in 4-bit nibbles (half bytes).

FIG. 3 shows an exemplary method 400 for performing a parallel lookup Galois field multiplication according to an embodiment of the present invention.

Referring to FIG. 3, in step 410, two lookup tables are built once: one lookup table for the low-order nibbles in each byte, and one lookup table for the high-order nibbles in each byte. Each lookup table contains 256 sets (one for each possible factor) of the 16 possible GF products of that factor and the 16 possible nibble values. Each lookup table is thus $256 \times 16 = 4096$ bytes, which is considerably smaller than the 65,536 bytes needed to store a complete one-byte multiplication table. In addition, PSHUFB does 16 separate table lookups at once, each for one nibble, so 8 PSHUFB instructions can be used to do all the table lookups for 64 bytes (128 nibbles).

Next, in step 420, the Parallel Lookup Multiplier is initialized for the next set of 64 bytes of operand data (such as original data or surviving original data). In order to save loading this data from memory on succeeding calls, the Parallel Lookup Multiplier dedicates four registers for this data, which are left intact upon exit of the Parallel Lookup Multiplier. This allows the same data to be called with different factors (such as processing the same data for another check drive).

Next in step 430, to process these 64 bytes of operand data, the Parallel Lookup Multiplier can be implemented with 2 MOVDQA (Move Double Quadword Aligned) instructions (from memory) to do the two table lookups and 4 MOVDQA instructions (register to register) to initialize registers (such as the output registers). These are followed in steps 440 and 450 by two nearly identical sets of 17 register-to-register instructions to carry out the multiplication 32 bytes at a time. Each such set starts (in step 440) with 5 more MOVDQA instructions for further initialization, followed by 2 PSRLW (Packed Shift Right Logical Word) instructions to realign the high-order nibbles for PSHUFB, and 4 PAND instructions to clear the high-order nibbles for PSHUFB. That is, two registers of byte operands are converted into four registers of nibble operands. Then, in step 450, 4 PSHUFB instructions are used to do the parallel table lookups, and 2 PXOR instructions to add the results of the multiplication on the two nibbles to the output registers.

Thus, the Parallel Lookup Multiplier uses 40 machine instructions to perform the parallel multiplication on 64 separate bytes, which is considerably better than the average 134 instructions for the Parallel Multiplier above, and only 10 times as many instructions as needed for the Parallel Adder. While some of the Parallel Lookup Multiplier's instructions are more complex than those of the Parallel Adder, much of this complexity can be concealed through the pipelined and/or concurrent execution of numerous such contiguous instructions (accessing different registers) on modern pipelined processors. For example, in exemplary implementations, the Parallel Lookup Multiplier has been timed at about 15 CPU clock cycles per 64 bytes processed per CPU core (about 0.36 clock cycles per instruction). In addition, the code footprint is practically nonexistent for the Parallel Lookup Multiplier (40 instructions) compared to that of the Parallel Multiplier (about 34,300 instructions),

US 10,003,358 B2

19

even when factoring the 8 KB needed for the two lookup tables in the Parallel Lookup Multiplier.

In addition, embodiments of the Parallel Lookup Multiplier can be passed 64 bytes of operand data (such as the next 64 bytes of surviving original data X to be processed) in four consecutive registers, whose contents can be preserved upon exiting the Parallel Lookup Multiplier (and all in the same 40 machine instructions) such that the Parallel Lookup Multiplier can be invoked again on the same 64 bytes of data without having to access main memory to reload the data. Through such a protocol, memory accesses can be minimized (or significantly reduced) for accessing the original data D during check data generation or the surviving original data X during lost data reconstruction.

Further embodiments of the present invention are directed towards sequencing this parallel multiplication (and other GF) operations. While the Parallel Lookup Multiplier processes a GF multiplication of 64 bytes of contiguous data times a specified factor, the calls to the Parallel Lookup Multiplier should be appropriately sequenced to provide efficient processing. One such sequencer (Sequencer 1), for example, can generate the check data J from the original data D, and is described further with respect to FIG. 4.

The parity drive does not need GF multiplication. The check data for the parity drive can be obtained, for example, by adding corresponding 64-byte chunks for each of the data drives to perform the parity operation. The Parallel Adder can do this using 4 instructions for every 64 bytes of data for each of the N data drives, or N/16 instructions per byte.

The M-1 non-parity check drives can invoke the Parallel Lookup Multiplier on each 64-byte chunk, using the appropriate factor for the particular combination of data drive and check drive. One consideration is how to handle the data access. Two possible ways are:

- 1) "column-by-column," i.e., 64 bytes for one data drive, followed by the next 64 bytes for that data drive, etc., and adding the products to the running total in memory (using the Parallel Adder) before moving onto the next row (data drive); and
- 2) "row-by-row," i.e., 64 bytes for one data drive, followed by the corresponding 64 bytes for the next data drive, etc., and keeping a running total using the Parallel Adder, then moving onto the next set of 64-byte chunks.

Column-by-column can be thought of as "constant factor, varying data," in that the (GF multiplication) factor usually remains the same between iterations while the (64-byte) data changes with each iteration. Conversely, row-by-row can be thought of as "constant data, varying factor," in that the data usually remains the same between iterations while the factor changes with each iteration.

Another consideration is how to handle the check drives. Two possible ways are:

- a) one at a time, i.e., generate all the check data for one check drive before moving onto the next check drive; and
- b) all at once, i.e., for each 64-byte chunk of original data, do all of the processing for each of the check drives before moving onto the next chunk of original data.

While each of these techniques performs the same basic operations (e.g., 40 instructions for every 64 bytes of data for each of the N data drives and M-1 non-parity check drives, or 5N(M-1)/8 instructions per byte for the Parallel Lookup Multiplier), empirical results show that combination (2)(b), that is, row-by-row data access on all of the check drives between data accesses performs best with the Parallel Lookup Multiplier. One reason may be that such an

20

approach appears to minimize the number of memory accesses (namely, one) to each chunk of the original data D to generate the check data J. This embodiment of Sequencer 1 is described in more detail with reference to FIG. 4.

FIG. 4 shows an exemplary method 500 for sequencing the Parallel Lookup Multiplier to perform the check data generation according to an embodiment of the present invention.

Referring to FIG. 4, in step 510, the Sequencer 1 is called. Sequencer 1 is called to process multiple 64-byte chunks of data for each of the blocks across a stripe of data. For instance, Sequencer 1 could be called to process 512 bytes from each block. If, for example, the block size L is 4096 bytes, then it would take eight such calls to Sequencer 1 to process the entire stripe. The other such seven calls to Sequencer 1 could be to different processing cores, for instance, to carry out the check data generation in parallel. The number of 64-byte chunks to process at a time could depend on factors such as cache dimensions, input/output data structure sizes, etc.

In step 520, the outer loop processes the next 64-byte chunk of data for each of the drives. In order to minimize the number of accesses of each data drive's 64-byte chunk of data from memory, the data is loaded only once and preserved across calls to the Parallel Lookup Multiplier. The first data drive is handled specially since the check data has to be initialized for each check drive. Using the first data drive to initialize the check data saves doing the initialization as a separate step followed by updating it with the first data drive's data. In addition to the first data drive, the first check drive is also handled specially since it is a parity drive, so its check data can be initialized to the first data drive's data directly without needing the Parallel Lookup Multiplier.

In step 530, the first middle loop is called, in which the remainder of the check drives (that is, the non-parity check drives) have their check data initialized by the first data drive's data. In this case, there is a corresponding factor (that varies with each check drive) that needs to be multiplied with each of the first data drive's data bytes. This is handled by calling the Parallel Lookup Multiplier for each non-parity check drive.

In step 540, the second middle loop is called, which processes the other data drives' corresponding 64-byte chunks of data. As with the first data drive, each of the other data drives is processed separately, loading the respective 64 bytes of data into four registers (preserved across calls to the Parallel Lookup Multiplier). In addition, since the first check drive is the parity drive, its check data can be updated by directly adding these 64 bytes to it (using the Parallel Adder) before handling the non-parity check drives.

In step 550, the inner loop is called for the next data drive. In the inner loop (as with the first middle loop), each of the non-parity check drives is associated with a corresponding factor for the particular data drive. The factor is multiplied with each of the next data drive's data bytes using the Parallel Lookup Multiplier, and the results added to the check drive's check data.

Another such sequencer (Sequencer 2) can be used to reconstruct the lost data from the surviving data (using Algorithm 2). While the same column-by-column and row-by-row data access approaches are possible, as well as the same choices for handling the check drives, Algorithm 2 adds another dimension of complexity because of the four separate steps and whether to: (i) do the steps completely serially or (ii) do some of the steps concurrently on the same data. For example, step 1 (surviving check data generation) and step 4 (lost check data regeneration) can be done

US 10,003,358 B2

21

concurrently on the same data to reduce or minimize the number of surviving original data accesses from memory.

Empirical results show that method (2)(b)(ii), that is, row-by-row data access on all of the check drives and for both surviving check data generation and lost check data regeneration between data accesses performs best with the Parallel Lookup Multiplier when reconstructing lost data using Algorithm 2. Again, this may be due to the apparent minimization of the number of memory accesses (namely, one) of each chunk of surviving original data X to reconstruct the lost data and the absence of memory accesses of reconstructed lost original data Y when regenerating the lost check data. This embodiment of Sequencer 1 is described in more detail with reference to FIGS. 5-7.

FIGS. 5-7 show an exemplary method 600 for sequencing the Parallel Lookup Multiplier to perform the lost data reconstruction according to an embodiment of the present invention.

Referring to FIG. 5, in step 610, the Sequencer 2 is called. Sequencer 2 has many similarities with the embodiment of Sequencer 1 illustrated in FIG. 4. For instance, Sequencer 2 processes the data drive data in 64-byte chunks like Sequencer 1. Sequencer 2 is more complex, however, in that only some of the data drive data is surviving; the rest has to be reconstructed. In addition, lost check data needs to be regenerated. Like Sequencer 1, Sequencer 2 does these operations in such a way as to minimize memory accesses of the data drive data (by loading the data once and calling the Parallel Lookup Multiplier multiple times). Assume for ease of description that there is at least one surviving data drive; the case of no surviving data drives is handled a little differently, but not significantly different. In addition, recall from above that the driving formula behind data reconstruction is $Y=B^{-1} \times (W-A \times X)$, where Y is the lost original data, B^{-1} is the solution matrix, W is the surviving check data, A is the partial check data encoding matrix (for the surviving check drives and the surviving data drives), and X is the surviving original data.

In step 620, the outer loop processes the next 64-byte chunk of data for each of the drives. Like Sequencer 1, the first surviving data drive is again handled specially since the partial check data $A \times X$ has to be initialized for each surviving check drive.

In step 630, the first middle loop is called, in which the partial check data $A \times X$ is initialized for each surviving check drive based on the first surviving data drive's 64 bytes of data. In this case, the Parallel Lookup Multiplier is called for each surviving check drive with the corresponding factor (from A) for the first surviving data drive.

In step 640, the second middle loop is called, in which the lost check data is initialized for each failed check drive. Using the same 64 bytes of the first surviving data drive (preserved across the calls to Parallel Lookup Multiplier in step 630), the Parallel Lookup Multiplier is again called, this time to initialize each of the failed check drive's check data to the corresponding component from the first surviving data drive. This completes the computations involving the first surviving data drive's 64 bytes of data, which were fetched with one access from main memory and preserved in the same four registers across steps 630 and 640.

Continuing with FIG. 6, in step 650, the third middle loop is called, which processes the other surviving data drives' corresponding 64-byte chunks of data. As with the first surviving data drive, each of the other surviving data drives is processed separately, loading the respective 64 bytes of data into four registers (preserved across calls to the Parallel Lookup Multiplier).

22

In step 660, the first inner loop is called, in which the partial check data $A \times X$ is updated for each surviving check drive based on the next surviving data drive's 64 bytes of data. In this case, the Parallel Lookup Multiplier is called for each surviving check drive with the corresponding factor (from A) for the next surviving data drive.

In step 670, the second inner loop is called, in which the lost check data is updated for each failed check drive. Using the same 64 bytes of the next surviving data drive (preserved across the calls to Parallel Lookup Multiplier in step 660), the Parallel Lookup Multiplier is again called, this time to update each of the failed check drive's check data by the corresponding component from the next surviving data drive. This completes the computations involving the next surviving data drive's 64 bytes of data, which were fetched with one access from main memory and preserved in the same four registers across steps 660 and 670.

Next, in step 680, the computation of the partial check data $A \times X$ is complete, so the surviving check data W is added to this result (recall that $W-A \times X$ is equivalent to $W+A \times X$ in binary Galois Field arithmetic). This is done by the fourth middle loop, which for each surviving check drive adds the corresponding 64-byte component of surviving check data W to the (surviving) partial check data $A \times X$ (using the Parallel Adder) to produce the (lost) partial check data $W-A \times X$.

Continuing with FIG. 7, in step 690, the fifth middle loop is called, which performs the two dimensional matrix multiplication $B^{-1} \times (W-A \times X)$ to produce the lost original data Y . The calculation is performed one row at a time, for a total of F rows, initializing the row to the first term of the corresponding linear combination of the solution matrix B^{-1} and the lost partial check data $W-A \times X$ (using the Parallel Lookup Multiplier).

In step 700, the third inner loop is called, which completes the remaining $F-1$ terms of the corresponding linear combination (using the Parallel Lookup Multiplier on each term) from the fifth middle loop in step 690 and updates the running calculation (using the Parallel Adder) of the next row of $B^{-1} \times (W-A \times X)$. This completes the next row (and reconstructs the corresponding failed data drive's lost data) of lost original data Y , which can then be stored at an appropriate location.

In step 710, the fourth inner loop is called, in which the lost check data is updated for each failed check drive by the newly reconstructed lost data for the next failed data drive. Using the same 64 bytes of the next reconstructed lost data (preserved across calls to the Parallel Lookup Multiplier), the Parallel Lookup Multiplier is called to update each of the failed check drives' check data by the corresponding component from the next failed data drive. This completes the computations involving the next failed data drive's 64 bytes of reconstructed data, which were performed as soon as the data was reconstructed and without being stored and retrieved from main memory.

Finally, in step 720, the sixth middle loop is called. The lost check data has been regenerated, so in this step, the newly regenerated check data is stored at an appropriate location (if desired).

Aspects of the present invention can be also realized in other environments, such as two-byte quantities, each such two-byte quantity capable of taking on $2^{16}=65,536$ possible values, by using similar constructs (scaled accordingly) to those presented here. Such extensions would be readily apparent to one of ordinary skill in the art, so their details will be omitted for brevity of description.

Exemplary techniques and methods for doing the Galois field manipulation and other mathematics behind RAID error correcting codes are described in Appendix A, which contains a paper “Information Dispersal Matrices for RAID Error Correcting Codes” prepared for the present application.

Multi-Core Considerations

What follows is an exemplary embodiment for optimizing or improving the performance of multi-core architecture systems when implementing the described erasure coding system routines. In multi-core architecture systems, each processor die is divided into multiple CPU cores, each with their own local caches, together with a memory (bus) interface and possible on-die cache to interface with a shared memory with other processor dies.

FIG. 8 illustrates a multi-core architecture system **100** having two processor dies **110** (namely, Die 0 and Die 1).

Referring to FIG. 8, each die **110** includes four central processing units (CPUs or cores) **120** each having a local level 1 (L1) cache. Each core **120** may have separate functional units, for example, an x86 execution unit (for traditional instructions) and a SSE execution unit (for software designed for the newer SSE instruction set). An example application of these function units is that the x86 execution unit can be used for the RAID control logic software while the SSE execution unit can be used for the GF operation software. Each die **110** also has a level 2 (L2) cache/memory bus interface **130** shared between the four cores **120**. Main memory **140**, in turn, is shared between the two dies **110**, and is connected to the input/output (I/O) controllers **150** that access external devices such as disk drives or other non-volatile storage devices via interfaces such as Peripheral Component Interconnect (PCI).

Redundant array of independent disks (RAID) controller processing can be described as a series of states or functions. These states may include: (1) Command Processing, to validate and schedule a host request (for example, to load or store data from disk storage); (2) Command Translation and Submission, to translate the host request into multiple disk requests and to pass the requests to the physical disks; (3) Error Correction, to generate check data and reconstruct lost data when some disks are not functioning correctly; and (4) Request Completion, to move data from internal buffers to requestor buffers. Note that the final state, Request Completion, may only be needed for a RAID controller that supports caching, and can be avoided in a cacheless design.

Parallelism is achieved in the embodiment of FIG. 8 by assigning different cores **120** to different tasks. For example, some of the cores **120** can be “command cores,” that is, assigned to the I/O operations, which includes reading and storing the data and check bytes to and from memory **140** and the disk drives via the I/O interface **150**. Others of the cores **120** can be “data cores,” and assigned to the GF operations, that is, generating the check data from the original data, reconstructing the lost data from the surviving data, etc., including the Parallel Lookup Multiplier and the sequencers described above. For example, in exemplary embodiments, a scheduler can be used to divide the original data *D* into corresponding portions of each block, which can then be processed independently by different cores **120** for applications such as check data generation and lost data reconstruction.

One of the benefits of this data core/command core subdivision of processing is ensuring that different code will be executed in different cores **120** (that is, command code in command cores, and data code in data cores). This improves the performance of the associated L1 cache in each core **120**, and avoids the “pollution” of these caches with code that is less frequently executed. In addition, empirical results show that the dies **110** perform best when only one core **120** on

each die **110** does the GF operations (i.e., Sequencer 1 or Sequencer 2, with corresponding calls to Parallel Lookup Multiplier) and the other cores **120** do the I/O operations. This helps localize the Parallel Lookup Multiplier code and associated data to a single core **120** and not compete with other cores **120**, while allowing the other cores **120** to keep the data moving between memory **140** and the disk drives via the I/O interface **150**.

Embodiments of the present invention yield scalable, high performance RAID systems capable of outperforming other systems, and at much lower cost, due to the use of high volume commodity components that are leveraged to achieve the result. This combination can be achieved by utilizing the mathematical techniques and code optimizations described elsewhere in this application with careful placement of the resulting code on specific processing cores. Embodiments can also be implemented on fewer resources, such as single-core dies and/or single-die systems, with decreased parallelism and performance optimization.

The process of subdividing and assigning individual cores **120** and/or dies **110** to inherently parallelizable tasks will result in a performance benefit. For example, on a Linux system, software may be organized into “threads,” and threads may be assigned to specific CPUs and memory systems via the `kthread_bind` function when the thread is created. Creating separate threads to process the GF arithmetic allows parallel computations to take place, which multiplies the performance of the system.

Further, creating multiple threads for command processing allows for fully overlapped execution of the command processing states. One way to accomplish this is to number each command, then use the arithmetic MOD function (`%` in C language) to choose a separate thread for each command. Another technique is to subdivide the data processing portion of each command into multiple components, and assign each component to a separate thread.

FIG. 9 shows an exemplary disk drive configuration **200** according to an embodiment of the present invention.

Referring to FIG. 9, eight disks are shown, though this number can vary in other embodiments. The disks are divided into three types: data drives **210**, parity drive **220**, and check drives **230**. The eight disks break down as three data drives **210**, one parity drive **220**, and four check drives **230** in the embodiment of FIG. 9.

Each of the data drives **210** is used to hold a portion of data. The data is distributed uniformly across the data drives **210** in stripes, such as 192 KB stripes. For example, the data for an application can be broken up into stripes of 192 KB, and each of the stripes in turn broken up into three 64 KB blocks, each of the three blocks being written to a different one of the three data drives **210**.

The parity drive **220** is a special type of check drive in that the encoding of its data is a simple summation (recall that this is exclusive OR in binary GF arithmetic) of the corresponding bytes of each of the three data drives **210**. That is, check data generation (Sequencer 1) or regeneration (Sequencer 2) can be performed for the parity drive **220** using the Parallel Adder (and not the Parallel Lookup Multiplier). Accordingly, the check data for the parity drive **220** is relatively straightforward to build. Likewise, when one of the data drives **210** no longer functions correctly, the parity drive **220** can be used to reconstruct the lost data by adding (same as subtracting in binary GF arithmetic) the corresponding bytes from each of the two remaining data drives **210**. Thus, a single drive failure of one of the data drives **210** is very straightforward to handle when the parity drive **220** is available (no Parallel Lookup Multiplier). Accordingly, the parity drive **220** can replace much of the GF multiplication operations with GF addition for both check data generation and lost data reconstruction.

Each of the check drives **230** contains a linear combination of the corresponding bytes of each of the data drives **210**. The linear combination is different for each check drive **230**, but in general is represented by a summation of different multiples of each of the corresponding bytes of the data drives **210** (again, all arithmetic being GF arithmetic). For example, for the first check drive **230**, each of the bytes of the first data drive **210** could be multiplied by 4, each of the bytes of the second data drive **210** by 3, and each of the bytes of the third data drive **210** by 6, then the corresponding products for each of the corresponding bytes could be added to produce the first check drive data. Similar linear combinations could be used to produce the check drive data for the other check drives **230**. The specifics of which multiples for which check drive are explained in Appendix A.

With the addition of the parity drive **220** and check drives **230**, eight drives are used in the RAID system **200** of FIG. 9. Accordingly, each 192 KB of original data is stored as 512 KB (i.e., eight blocks of 64 KB) of (original plus check) data. Such a system **200**, however, is capable of recovering all of the original data provided any three of these eight drives survive. That is, the system **200** can withstand a concurrent failure of up to any five drives and still preserve all of the original data.

Exemplary Routines to Implement an Embodiment

The error correcting code (ECC) portion of an exemplary embodiment of the present invention may be written in software as, for example, four functions, which could be named as ECCInitialize, ECCSolve, ECCGenerate, and ECCRegenerate. The main functions that perform work are ECCGenerate and ECCRegenerate. ECCGenerate generates check codes for data that are used to recover data when a drive suffers an outage (that is, ECCGenerate generates the check data J from the original data D using Sequencer 1). ECCRegenerate uses these check codes and the remaining data to recover data after such an outage (that is, ECCRegenerate uses the surviving check data W, the surviving original data X, and Sequencer 2 to reconstruct the lost original data Y while also regenerating any of the lost check data). Prior to calling either of these functions, ECCSolve is called to compute the constants used for a particular configuration of data drives, check drives, and failed drives (for example, ECCSolve builds the solution matrix B^{-1} together with the lists of surviving and failed data and check drives). Prior to calling ECCSolve, ECCInitialize is called to generate constant tables used by all of the other functions (for example, ECCInitialize builds the IDM E and the two lookup tables for the Parallel Lookup Multiplier).

ECCInitialize

The function ECCInitialize creates constant tables that are used by all subsequent functions. It is called once at program initialization time. By copying or precomputing these values up front, these constant tables can be used to replace more time-consuming operations with simple table look-ups (such as for the Parallel Lookup Multiplier). For example, four tables useful for speeding up the GF arithmetic include:

1. mvct—an array of constants used to perform GF multiplication with the PSHUFB instruction that operates on SSE registers (that is, the Parallel Lookup Multiplier).

2. mast—contains the master encoding matrix S (or the Information Dispersal Matrix (IDM) E, as described in Appendix A), or at least the nontrivial portion, such as the check drive encoding matrix H

3. mul_tab—contains the results of all possible GF multiplication operations of any two operands (for example, $256 \times 256 = 65,536$ bytes for all of the possible products of two different one-byte quantities)

4. div_tab—contains the results of all possible GF division operations of any two operands (can be similar in size to mul_tab)

ECCSolve

The function ECCSolve creates constant tables that are used to compute a solution for a particular configuration of data drives, check drives, and failed drives. It is called prior to using the functions ECCGenerate or ECCRegenerate. It allows the user to identify a particular case of failure by describing the logical configuration of data drives, check drives, and failed drives. It returns the constants, tables, and lists used to either generate check codes or regenerate data. For example, it can return the matrix B that needs to be inverted as well as the inverted matrix B^{-1} (i.e., the solution matrix).

ECCGenerate

The function ECCGenerate is used to generate check codes (that is, the check data matrix J) for a particular configuration of data drives and check drives, using Sequencer 1 and the Parallel Lookup Multiplier as described above. Prior to calling ECCGenerate, ECCSolve is called to compute the appropriate constants for the particular configuration of data drives and check drives, as well as the solution matrix B^{-1} .

ECCRegenerate

The function ECCRegenerate is used to regenerate data vectors and check code vectors for a particular configuration of data drives and check drives (that is, reconstructing the original data matrix D from the surviving data matrix X and the surviving check matrix W, as well as regenerating the lost check data from the restored original data), this time using Sequencer 2 and the Parallel Lookup Multiplier as described above. Prior to calling ECCRegenerate, ECCSolve is called to compute the appropriate constants for the particular configuration of data drives, check drives, and failed drives, as well as the solution matrix B^{-1} .

Exemplary Implementation Details

As discussed in Appendix A, there are two significant sources of computational overhead in erasure code processing (such as an erasure coding system used in RAID processing): the computation of the solution matrix B^{-1} for a given failure scenario, and the byte-level processing of encoding the check data J and reconstructing the lost data after a lost packet (e.g., data drive failure). By reducing the solution matrix B^{-1} to a matrix inversion of a $F \times F$ matrix, where F is the number of lost packets (e.g., failed drives), that portion of the computational overhead is for all intents and purposes negligible compared to the megabytes (MB), gigabytes (GB), and possibly terabytes (TB) of data that needs to be encoded into check data or reconstructed from the surviving original and check data. Accordingly, the remainder of this section will be devoted to the byte-level encoding and regenerating processing.

As already mentioned, certain practical simplifications can be assumed for most implementations. By using a Galois field of 256 entries, byte-level processing can be used for all of the GF arithmetic. Using the master encoding matrix S described in Appendix A, any combination of up to 127 data drives, 1 parity drive, and 128 check drives can be supported with such a Galois field. While, in general, any combination of data drives and check drives that adds up to 256 total drives is possible, not all combinations provide a parity drive when computed directly. Using the master encoding matrix S, on the other hand, allows all such combinations (including a parity drive) to be built (or simply indexed) from the same such matrix. That is, the appropriate sub-matrix (including the parity drive) can be used for configurations of less than the maximum number of drives.

In addition, using the master encoding matrix S permits further data drives and/or check drives can be added without requiring the recomputing of the IDM E (unlike other

US 10,003,358 B2

27

proposed solutions, which recompute E for every change of N or M). Rather, additional indexing of rows and/or columns of the master encoding matrix S will suffice. As discussed above, the use of the parity drive can eliminate or significantly reduce the somewhat complex GF multiplication operations associated with the other check drives and replaces them with simple GF addition (bitwise exclusive OR in binary Galois fields) operations. It should be noted that master encoding matrices with the above properties are possible for any power-of-two number of drives $2^P = N_{max} + M_{max}$, where the maximum number of data drives N_{max} is one less than a power of two (e.g., $N_{max} = 127$ or 63) and the maximum number of check drives M_{max} (including the parity drive) is $2^P - N_{max}$.

As discussed earlier, in an exemplary embodiment of the present invention, a modern x86 architecture is used (being readily available and inexpensive). In particular, this architecture supports 16 XMM registers and the SSE instructions. Each XMM register is 128 bits and is available for special purpose processing with the SSE instructions. Each of these XMM registers holds 16 bytes (8-bit), so four such registers can be used to store 64 bytes of data. Thus, by using SSE instructions (some of which work on different operand sizes, for example, treating each of the XMM registers as containing 16 one-byte operands), 64 bytes of data can be operated at a time using four consecutive SSE instructions (e.g., fetching from memory, storing into memory, zeroing, adding, multiplying), the remaining registers being used for intermediate results and temporary storage. With such an architecture, several routines are useful for optimizing the byte-level performance, including the Parallel Lookup Multiplier, Sequencer 1, and Sequencer 2 discussed above.

While the above description contains many specific embodiments of the invention, these should not be construed as limitations on the scope of the invention, but rather as examples of specific embodiments thereof. Accordingly, the scope of the invention should be determined not by the embodiments illustrated, but by the appended claims and their equivalents.

Glossary of Some Variables

A encoding matrix (F×K), sub-matrix of T
 B encoding matrix (F×F), sub-matrix of T
 B⁻¹ solution matrix (F×F)
 C encoded data matrix

$$((N + M) \times L) = \begin{bmatrix} D \\ J \end{bmatrix}$$

C' surviving encoded data matrix

$$(N \times L) = \begin{bmatrix} X \\ W \end{bmatrix}$$

D original data matrix (N×L)
 D' permuted original data matrix

$$(N \times L) = \begin{bmatrix} X \\ Y \end{bmatrix}$$

E information dispersal matrix

$$(IDM)((N + M) \times N) = \begin{bmatrix} I_N \\ H \end{bmatrix}$$

28

F number of failed data drives
 G number of failed check drives
 H check drive encoding matrix (M×N)
 I identity matrix ($I_K = K \times K$ identity matrix, $I_N = N \times N$ identity matrix)
 J encoded check data matrix (M×L)
 K number of surviving data drives = N - F
 L data block size (elements or bytes)
 M number of check drives
 M_{max} maximum value of M
 N number of data drives
 N_{max} maximum value of N
 O zero matrix (K×F), sub-matrix of T
 S master encoding matrix ($(M_{max} + N_{max}) \times N_{max}$)
 T transformed IDM

$$(N \times N) = \begin{bmatrix} I_K & O \\ A & B \end{bmatrix}$$

W surviving check data matrix (F×L)
 X surviving original data matrix (K×L)
 Y lost original data matrix (F×L)

What is claimed is:

1. A system adapted to use accelerated error-correcting code (ECC) processing to improve the storage and retrieval of digital data distributed across a plurality of drives, comprising:

at least one processor comprising at least one single-instruction-multiple-data (SIMD) central processing unit (CPU) core that executes SIMD instructions and loads original data from a main memory and stores check data to the main memory, the SIMD CPU core comprising at least 16 vector registers, each of the vector registers storing at least 16 bytes;

at least one system drive comprising at least one non-volatile storage medium that stores the SIMD instructions;

a plurality of data drives each comprising at least one non-volatile storage medium that stores at least one block of the original data, the at least one block comprising at least 512 bytes;

more than two check drives each comprising at least one non-volatile storage medium that stores at least one block of the check data; and

at least one input/output (I/O) controller that stores the at least one block of the check data from the main memory to the check drives,

wherein the processor, the SIMD instructions, the non-volatile storage media, and the I/O controller are configured to implement an erasure coding system comprising:

a data matrix comprising at least one vector and comprising a plurality of rows of at least one block of the original data in the main memory, each of the rows being stored on a different one of the data drives;

a check matrix comprising more than two rows of the at least one block of the check data in the main memory, each of the rows being stored on a different one of the check drives, one of the rows comprising a parity row comprising the Galois Field (GF) summation of all of the rows of the data matrix;

a thread that executes on the SIMD CPU core and comprising:

at least one parallel multiplier that multiplies the at least one vector of the data matrix by a single

US 10,003,358 B2

29

factor to compute parallel multiplier results comprising at least one vector;
 at least one parallel adder that adds the at least one vector of the parallel multiplier results and computes a running total; and
 a sequencer wherein the sequencer orders load operations of the original data into at least one of the vector registers and computes the check data with the parallel lookup multiplier and the parallel adder, and stores the computed check data from the vector registers to the main memory.

2. The system of claim 1, wherein:
 the processor comprises a first CPU core and a second CPU core;
 the thread comprises a plurality of threads comprising a first thread group and a second thread group; and
 the erasure coding system further comprises a scheduler for performing data operations to generate the check data and, concurrently, performing I/O operations using the I/O controller by:
 assigning the data operations to the first thread group, and not assigning the I/O operations to the first thread group;
 assigning the I/O operations to the second thread group and not assigning the data operations to the second thread group;
 assigning the first thread group to the first CPU core; assigning the second thread group to the second CPU core; and
 concurrently executing the first thread group on the first CPU core and the second thread group on the second CPU core to concurrently generate the check data and perform the I/O operations.

3. The system of claim 1, wherein the sequencer loads each entry of the data matrix from the main memory into a vector register at most once while generating the check data.

4. The system of claim 1, wherein the at least one processor is an x86 architecture processor.

5. The system of claim 1, wherein the erasure coding system further comprises:
 an encoding matrix comprising more than two but not more than 254 rows and more than one but not more than 253 columns of factors in the main memory, wherein each of the entries of one of the rows of the encoding matrix comprises a multiplicative identity factor, the factors being for encoding the original data into the check data.

6. The system of claim 5, wherein the at least one parallel multiplier multiplies the at least one vector of the data matrix in units of at least 64 bytes.

7. The system of claim 5, wherein the data matrix comprises a first number of rows and the data drives comprise the first number of data drives,
 wherein the check matrix comprises a second number of rows and the check drives comprise the second number of check drives, and
 wherein the encoding matrix comprises a plurality of first factors in the second number of rows and the first number of columns.

8. The system of claim 7, wherein the encoding matrix further comprises a third number of columns and a plurality of second factors in the third number of columns,
 wherein the data drives further comprise the third number of data drives, and
 wherein the first factors are independent of the third number.

30

9. The system of claim 7, wherein the encoding matrix further comprises a fourth number of rows and a plurality of third factors in the fourth number of rows,
 wherein the check drives further comprise the fourth number of check drives, and
 wherein the first factors are independent of the fourth number.

10. The system of claim 5, wherein the multiplicative identity factor is 1.

11. The system of claim 1, wherein the at least one parallel multiplier multiplies the at least one vector of the data matrix by the single factor in the encoding matrix at a rate of less than about 2 machine instructions per byte of the data matrix.

12. A system adapted to use accelerated error-correcting code (ECC) processing to improve the storage and retrieval of digital data distributed across a plurality of drives, comprising:
 at least one processor comprising at least one single-instruction-multiple-data (SIMD) central processing unit (CPU) core that executes SIMD instructions and loads surviving original data and surviving check data from a main memory and stores lost original data to the main memory, the SIMD CPU core comprising at least 16 vector registers, each of the vector registers storing at least 16 bytes;
 at least one system drive comprising at least one non-volatile storage medium that stores the SIMD instructions;
 a plurality of data drives each comprising at least one non-volatile storage medium that stores at least one block of the original data, the at least one block comprising at least 512 bytes;
 more than two check drives each comprising at least one non-volatile storage medium that stores at least one block of the check data; and
 at least one input/output (I/O) controller that reads at least one block of the check data from the check drives and stores the at least one block of the check data to the main memory,
 wherein the processor, the SIMD instructions, the non-volatile storage media and the I/O controller implement the accelerated ECC processing, comprising:
 a surviving data matrix comprising at least one vector and comprising at least one row of at least one block of the surviving original data in the main memory, each row of the at least one row being stored on a different one of the data drives, and a lost data matrix comprising at least one block of the lost original data in the main memory;
 a surviving check matrix comprising at least one row of at least one block of the surviving check data in the main memory, each row of the at least one row being stored on a different one of the check drives;
 a solution matrix that holds factors in the main memory, the factors of the solution matrix being for decoding the surviving original data and the surviving check data into the lost original data;
 and
 a thread that executes on the SIMD CPU core and comprising:
 at least one parallel multiplier that multiplies the at least one vector of the surviving data matrix by a single factor in the solution matrix to compute parallel multiplier results comprising at least one vector;

US 10,003,358 B2

31

at least one parallel adder that adds the at least one vector of the parallel multiplier results and computes a running total; and

a sequencer wherein the sequencer:

orders load operations of the surviving original data into at least one of the vector registers and load operations of the surviving check data into at least one of the vector registers; computes the lost original data with the parallel multiplier and the parallel adder; and stores the computed lost original data from the vector registers to the lost data matrix.

13. The system of claim 12, wherein:

the processing core comprises a first CPU core and a second CPU core;

the thread comprises a plurality of threads comprising a first thread group and a second thread group; and

the erasure coding system further comprises a scheduler for performing data operations to regenerate the lost original data and, concurrently, performing I/O operations using the I/O controller by:

assigning the data operations to the first thread group, and not assigning the I/O operations to the first thread group;

assigning the I/O operations to the second thread group, and not assigning the data operations to the second thread group;

assigning the first thread group to the first CPU core; assigning the second thread group to the second CPU core; and

concurrently executing the first thread group on the first CPU core and the second thread group on the second CPU core to concurrently regenerate the lost original data and perform the I/O operations.

14. The system of claim 12, wherein the sequencer loads each entry of the surviving original data from the main memory into a vector register at most once while regenerating the lost original data.

15. The system of claim 12, wherein the at least one parallel multiplier multiplies the at least one vector of the surviving data matrix in units of at least 64 bytes.

16. The system of claim 12, wherein the processor is an x86 architecture processor.

17. The system of claim 12, wherein the solution matrix comprises an inverted sub-matrix of an encoding matrix and wherein each of entries of one of the rows of the encoding matrix comprises a multiplicative identity factor, the factors of the encoding matrix being for encoding the original data into the check data.

18. The system of claim 17, wherein the multiplicative identity factor is 1.

19. The system of claim 12, wherein the at least one parallel multiplier multiplies the at least one vector of the surviving data matrix by the single factor in the solution matrix at a rate of less than about 2 machine instructions per byte of the surviving data matrix.

20. A method for accelerated error-correcting code (ECC) processing to improve the storage and retrieval of digital data distributed across a plurality of drives using a computing system, the computing system comprising:

at least one processor comprising at least one single-instruction-multiple-data (SIMD) central processing unit (CPU) core that executes a computer program including SIMD computer instructions and loads original data from a main memory and stores check data to

32

the main memory, the SIMD CPU core comprising at least 16 vector registers, each of the vector registers storing at least 16 bytes;

at least one system drive comprising at least one non-volatile storage medium that stores the SIMD computer instructions;

a plurality of data drives each comprising at least one non-volatile storage medium that stores at least one block of the original data, the at least one block comprising at least 512 bytes;

more than two check drives each comprising at least one non-volatile storage medium that stores at least one block of the check data; and

at least one input/output (I/O) controller that stores the at least one block of the check data from the main memory to the check drives, the method comprising: accessing the SIMD instructions from the system drive; executing the SIMD instructions on the SIMD CPU core;

arranging the original data as a data matrix comprising at least one vector and comprising a plurality of rows of at least one block of the original data in the main memory, each of the rows being stored on a different one of the data drives;

arranging the check data as a check matrix comprising more than two rows of the at least one block of the check data in the main memory, each of the rows being stored on a different one of the check drives, one of the rows comprising a parity row comprising the Galois Field (GF) summation of all of the rows of the data matrix; and

encoding the original data into the check data using: at least one parallel multiplier that multiplies the at least one vector of the data matrix by a single factor to compute parallel multiplier results comprising at least one vector; and

at least one parallel adder that adds the at least one vector of the parallel multiplier results and computes a running total,

the encoding of the check data comprising:

loading the original data into at least one of the vector registers;

computing the check data with the parallel multiplier and the parallel adder; and

storing the computed check data from the vector registers into the main memory.

21. The method of claim 20, wherein:

the processor comprises a first CPU core and a second CPU core;

the executing of the SIMD instructions comprises executing the SIMD instructions on the first CPU core to perform data operations to generate the check data and, concurrently, to perform I/O operations on the second CPU core to control the I/O controller;

the method further comprises scheduling the data operations concurrently with the I/O operations by:

assigning the data operations to the first CPU core, and not assigning the I/O operations to the first CPU core; and

assigning the I/O operations to the second CPU core and not assigning the data operations to the second CPU core.

22. The method of claim 20, further comprising loading each entry of the data matrix from the main memory into a vector register at most once while generating the check data.

23. The method of claim 20, wherein the processor is an x86 architecture processor.

US 10,003,358 B2

33

24. The method of claim 20, further comprising:
arranging factors as an encoding matrix comprising more
than two but not more than 254 rows and more than one
but not more than 253 columns of factors in the main
memory, wherein each of the entries of one of the rows
of the encoding matrix comprises a multiplicative identity
factor, the factors being for encoding the original
data into the check data. 5

25. The method of claim 24, wherein the at least one
parallel multiplier multiplies the at least one vector of the
data matrix in units of at least 64 bytes. 10

26. The method of claim 24, wherein the data matrix
comprises a first number of rows and the data drives
comprise the first number of data drives, 15
wherein the check matrix comprises a second number of
rows and the check drives comprise the second number
of check drives, and
wherein the encoding matrix comprises a plurality of first
factors in the second number of rows and the first
number of columns. 20

27. The method of claim 26, further comprising:
adding a third number of data drives to the data drives by
expanding the encoding matrix to further comprise the
third number of columns and a plurality of second
factors in the third number of columns, 25
wherein the first factors are independent of the third
number.

28. The method of claim 26, further comprising:
adding a fourth number of check drives to the check
drives by expanding the encoding matrix to further
comprise the fourth number of rows and a plurality of
third factors in the fourth number of rows, 30
wherein the first factors are independent of the fourth
number. 35

29. The method of claim 24, wherein the at least one
parallel multiplier multiplies the at least one vector of the
data matrix by the single factor in the encoding matrix at a
rate of less than about 2 machine instructions per byte of the
data matrix. 40

30. The method of claim 20, wherein the multiplicative
identity factor is 1.

31. A method for accelerated error-correcting code (ECC)
processing to improve the storage and retrieval of digital
data distributed across a plurality of drives using a comput-
ing system, the computing system comprising:
at least one processor comprising at least one single-
instruction-multiple-data (SIMD) central processing
unit (CPU) core that executes a computer program
including SIMD instructions and loads surviving original
data and surviving check data from a main memory and
stores lost original data to the main memory, the
SIMD CPU core comprising at least 16 vector registers,
each of the vector registers storing at least 16 bytes; 55
at least one system drive comprising at least one non-
volatile storage medium that stores the SIMD instruc-
tions;
a plurality of data drives each comprising at least one
non-volatile storage medium that stores at least one
block of the original data, the at least one block
comprising at least 512 bytes; 60
more than two check drives each comprising at least one
non-volatile storage medium that stores at least one
block of the check data; 65
at least one input/output (I/O) controller that reads at least
one block of the surviving check data from the check

34

drives and stores the at least one block of the surviving
check data to the main memory, the method compris-
ing:
accessing the SIMD instructions from the system drive;
executing the SIMD instructions on the SIMD CPU
core;
arranging the original data as a surviving data matrix
comprising at least one vector and comprising at
least one row of at least one block of the surviving
original data in the main memory, each row of the at
least one row being stored on a different one of the
data drives, and a lost data matrix comprising at least
one block of the lost original data in the main
memory;
arranging factors as a solution matrix that holds the
factors in the main memory, the factors being for
decoding the surviving original data and the surviv-
ing check data into the lost original data, the surviv-
ing check data being arranged as a surviving check
matrix comprising at least one row of at least one
block of the surviving check data in the main
memory, each row of the at least one row being
stored on a different one of the check drives;
decoding the surviving check data into the lost original
data using:
at least one parallel multiplier that multiplies the at
least one vector of the surviving data matrix by a
single factor in the solution matrix to compute par-
allel multiplier results comprising at least one vector;
and
at least one parallel adder that adds the at least one
vector of the parallel multiplier results and computes
a running total,
the decoding the surviving check data into the lost
original data comprising:
loading the surviving original data into at least one of
the vector registers;
loading the surviving check data into at least one of
the vector registers;
computing the lost original data with the parallel
multiplier and the parallel adder; and
storing the computed lost original data from the
vector registers into the lost data matrix.

32. The method of claim 31, wherein:
the processor comprises a first CPU core and a second
CPU core;
the executing of the SIMD instructions comprises execut-
ing the SIMD instructions on the first CPU core to
perform data operations to reconstruct the lost original
data and, concurrently, to perform I/O operations on the
second CPU core to control the I/O controller;
the method further comprises scheduling the data opera-
tions to be performed concurrently with the I/O opera-
tions by:
assigning the data operations to the first CPU core, and
not assigning the I/O operations to the first CPU
core; and
assigning the I/O operations to the second CPU core,
and not assigning the data operations to the first CPU
core.

33. The method of claim 31, further comprising loading
each entry of the surviving original data from the main
memory into a vector register at most once while regener-
ating the lost original data.

34. The method of claim 31, wherein the at least one
parallel multiplier multiplies the at least one vector of the
surviving data matrix in units of at least 64 bytes.

US 10,003,358 B2

35

35. The method of claim 31, wherein the processor is an x86 architecture processor.

36. The method of claim 31, wherein the solution matrix comprises an inverted sub-matrix of an encoding matrix and wherein each of entries of one of the rows of the encoding matrix comprises a multiplicative identity factor, the factors of the encoding matrix being for encoding the original data into the check data.

37. The method of claim 36, wherein the multiplicative identity factor is 1.

38. The method of claim 31, wherein the at least one parallel multiplier multiplies the at least one vector of the surviving data matrix by the single factor in the solution matrix at a rate of less than about 2 machine instructions per byte of the surviving data matrix.

39. A system drive comprising at least one non-transitory computer-readable storage medium containing a computer program comprising a plurality of computer instructions that, when executed by a computing system, cause the computing system to perform accelerated error-correcting code (ECC) processing that improves the storage and retrieval of digital data distributed across a plurality of drives, the computing system comprising:

at least one processor comprising at least one single-instruction-multiple-data (SIMD) central processing unit (CPU) core that executes SIMD instructions and loads original data from a main memory and stores check data to the main memory, the SIMD CPU core comprising at least 16 vector registers, each of the vector registers storing at least 16 bytes;

a plurality of data drives each comprising at least one non-volatile storage medium that stores at least one block of the original data, the at least one block comprising at least 512 bytes;

more than two check drives each comprising at least one non-volatile storage medium that stores at least one block of the check data; and

at least one input/output (I/O) controller that stores the at least one block of the check data from the main memory to the check drives,

the computer instructions implementing protection of the original data in the main memory when executed on the computing system by:

arranging the original data as a data matrix comprising at least one vector and comprising a plurality of rows of at least one block of the original data in the main memory, each of the rows being stored on a different one of the data drives;

arranging the check data as a check matrix comprising more than two rows of the at least one block of the check data in the main memory, each of the rows being stored on a different one of the check drives, one of the rows comprising a parity row comprising the Galois Field (GF) summation of all of the rows of the data matrix; and

encoding the original data into the check data using:

at least one parallel multiplier that multiplies the at least one vector of the data matrix by a single factor in the encoding matrix to compute parallel multiplier results comprising at least one vector; and

at least one parallel adder that adds the at least one vector of the parallel multiplier results and computes a running total,

the encoding the original data into the check data comprising:

36

loading the original data into at least one of the vector registers;

computing the check data with the parallel multiplier and the parallel adder; and

storing the computed check data from the vector registers into the main memory.

40. The system drive of claim 39, wherein: the processor comprises a first CPU core and a second CPU core;

the executing of the computer instructions comprises executing the computer instructions on the first CPU core to perform data operations to generate the check data and, concurrently, to perform I/O operations on the second CPU core to control the I/O controller;

the computer instructions implementing the protection of the original data comprise instructions that schedule the data operations to be performed concurrently with the I/O operations by:

assigning the data operations to the first CPU core, and not assigning the I/O operations to the first CPU core; and

assigning the I/O operations to the second CPU core and not assigning the data operations to the second CPU core.

41. The system drive of claim 39, wherein the computer instructions further comprise computer instructions that, when executed by the computing system, cause the computing system to load each entry of the data matrix from the main memory into a vector register at most once while generating the check data.

42. The system drive of claim 39, wherein the processor is an x86 architecture processor.

43. The system drive of claim 39, wherein the computer instructions implementing the protection of the original data comprise instructions to:

arrange factors as an encoding matrix comprising more than two but not more than 254 rows and more than one but not more than 253 columns of factors in the main memory, wherein each of the entries of one of the rows of the encoding matrix comprises a multiplicative identity factor, the factors being for encoding the original data into the check data.

44. The system drive of claim 43, wherein the at least one parallel multiplier multiplies the at least one vector of the data matrix in units of at least 64 bytes.

45. The system drive of claim 43, wherein the data matrix comprises a first number of rows and the data drives comprise the first number of data drives,

wherein the check matrix comprises a second number of rows and the check drives comprise the second number of check drives, and

wherein the encoding matrix comprises a plurality of first factors in the second number of rows and the first number of columns.

46. The system drive of claim 45, wherein the computer instructions further comprise instructions that, when executed on the computing system, cause the computing system to:

add a third number of data drives to the data drives by expanding the encoding matrix to further comprise the third number of columns and a plurality of second factors in the third number of columns,

wherein the first factors are independent of the third number.

37

47. The system drive of claim 45, wherein the computer instructions further comprise instructions that, when executed on the computing system, cause the computing system to:

add a fourth number of check drives to the check drives 5
 by expanding the encoding matrix to further comprise
 the fourth number of rows and a plurality of third
 factors in the fourth number of rows,
 wherein the first factors are independent of the fourth
 number. 10

48. The system drive of claim 43, wherein the multiplicative identity factor is 1.

49. The system drive of claim 43, wherein the at least one parallel multiplier multiplies the at least one vector of the data matrix by the single factor in the encoding matrix at a rate of less than about 2 machine instructions per byte of the data matrix. 15

50. A system drive comprising at least one non-transitory computer-readable storage medium containing a computer program comprising a plurality of computer instructions that, when executed by a computing system, cause the computing system to perform accelerated error-correcting code (ECC) processing that improves the storage and retrieval of digital data distributed across a plurality of drives, the en-a-computing system comprising: 20

at least one processor comprising at least one single-instruction-multiple-data (SIMD) central processing unit (CPU) core that executes SIMD instructions and loads surviving original data and surviving check data from a main memory and stores lost original data to the main memory, the SIMD CPU core comprising at least 16 vector registers, each of the vector registers storing at least 16 bytes; 30

a plurality of data drives each comprising at least one non-volatile storage medium that stores at least one block of the original data, the at least one block comprising at least 512 bytes; 35

more than two check drives each comprising at least one non-volatile storage medium that stores at least one block of the check data; 40

at least one input/output (I/O) controller that reads at least one block of the check data from the check drives and stores the at least one block of the check data to the main memory; 45

the computer instructions implementing protection of the original data in the main memory when executed on the computing system by:

arranging the surviving original data as a surviving data matrix comprising at least one vector and comprising at least one row of at least one block of the surviving original data in the main memory, each row of the at least one row being stored on a different one of the data drives, and a lost data matrix comprising at least one block of the lost original data in the main memory; 55

arranging factors as a solution matrix that holds the factors in the main memory, the factors being for decoding the surviving original data and the surviving check data into the lost original data, the surviving check data arranged as a surviving check matrix comprising at least one row of at least one block of the surviving check data in the main memory, each row of the at least one row being stored on a different one of the check drives; and 60

38

decoding the surviving check data into the lost original data using:

at least one parallel multiplier that multiplies the at least one vector of the surviving data matrix by a single factor in the solution matrix to compute parallel multiplier results comprising at least one vector; and

at least one parallel adder that adds the at least one vector of the parallel multiplier results and computes a running total,

decoding the surviving check data into the lost original data comprising:

loading the surviving original data into at least one of the vector registers;

loading the surviving check data into at least one of the vector registers;

computing the lost original data with the parallel multiplier and the parallel adder; and

storing the computed lost original data from the vector registers into the lost data matrix.

51. The system drive of claim 50, wherein: the processor comprises a first CPU core and a second CPU core;

the executing of the computer instructions comprises executing the computer instructions on the first CPU core to perform data operations to reconstruct the lost original data and, concurrently, to perform I/O operations on the second CPU core to control the I/O controller;

the computer instructions further comprise instructions that schedule the data operations to be performed concurrently with the I/O operations by:

assigning the data operations to the first CPU core, and not assigning the I/O operations to the first CPU core; and

assigning the I/O operations to the second CPU core, and not assigning the data operations to the first CPU core.

52. The system drive of claim 50, wherein the computer instructions further comprise computer instructions that, when executed on the computing system, cause the computing system to load each entry of the surviving original data from the main memory into a vector register at most once while regenerating the lost original data.

53. The system drive of claim 50, wherein the at least one parallel multiplier multiplies the at least one vector of the surviving data matrix in units of at least 64 bytes.

54. The system drive of claim 50, wherein the processor is an x86 architecture processor.

55. The system drive of claim 50 wherein the solution matrix comprises an inverted sub-matrix of an encoding matrix and wherein each of entries of one of the rows of the encoding matrix comprises a multiplicative identity factor, the factors of the encoding matrix being for encoding the original data into the check data.

56. The system drive of claim 55, wherein the multiplicative identity factor is 1.

57. The system drive of claim 50, wherein the at least one parallel multiplier multiplies the at least one vector of the surviving data matrix by the single factor in the solution matrix at a rate of less than about 2 machine instructions per byte of the surviving data matrix.

* * * * *

UNITED STATES PATENT AND TRADEMARK OFFICE
CERTIFICATE OF CORRECTION

PATENT NO. : 10,003,358 B2
APPLICATION NO. : 15/201196
DATED : June 19, 2018
INVENTOR(S) : Michael H. Anderson

Page 1 of 1

It is certified that error appears in the above-identified patent and that said Letters Patent is hereby corrected as shown below:

In the Claims

Column 28, Line 63, Claim 1, after “matric;” insert -- and --

Column 29, Line 9, Claim 1, after “parallel” delete “lookup”

Column 37, Line 25, Claim 50, delete “en-a-computing” and insert -- computing --

Column 38, Line 11, Claim 50, before “decoding” insert -- the --

Signed and Sealed this
Eleventh Day of December, 2018



Andrei Iancu
Director of the United States Patent and Trademark Office

EXHIBIT E

PATENT

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

I hereby certify that this correspondence is being EFS-Web transmitted to the United States Patent and Trademark Office on February 23, 2021 at or before 11:59 p.m. Pacific Time under the Rules of 37 CFR § 1.8.

/Jennifer Guerra/
Jennifer Guerra

Inventor(s) : Michael H. Anderson et al. Confirmation No. 1895
Assignee : Streamscale, Inc.
Patent No. : 10,003,358
Issued : June 19, 2018
Application No. : 15/201,196
Filed : July 1, 2016
Title : ACCELERATED ERASURE CODING SYSTEM AND METHOD
Docket No. : 124596/411563-00010

**PETITION FOR CORRECTION OF INVENTORSHIP
UNDER 37 CFR § 1.324**

Mail Stop Petition
Commissioner for Patents
P.O. Box 1450
Alexandria, VA 22313-1450

Post Office Box 29001
Glendale, CA 91209-9001
February 23, 2021

Commissioner:

Pursuant to 37 C.F.R. §1.324, Applicant respectfully requests the correction of inventorship for the above issued patent to include inventor Sarah Mann. Ms. Mann was not named as an inventor through error.

Enclosed are:

(1) Statement of Sarah Mann in Support of Petition for Correction of Inventorship Pursuant to 37 C.F.R. §1.324;

(2) Statement of Michael Anderson in Support of Petition for Correction of Inventorship Pursuant to 37 C.F.R. §1.324;

Patent No. 10,003,358

(3) Statement of Assignee, Streamscale, Inc., in Support of Petition for Correction of Inventorship Pursuant to 37 C.F.R. §1.324 and Complying with 37 C.F.R. §3.73(c);

(4) Executed Inventors Declaration and Assignment document signed by Sarah Mann; and

(5) Application Data Sheet.

The required fee of \$160.00 as required by §1.20(b). The Commissioner is hereby authorized to charge any fees as required by this petition to Deposit Account No. 03-1728. Please show our docket number with any charge or credit to our deposit account.

Respectfully submitted,

LEWIS ROCA ROTHGERBER CHRISTIE LLP

By /David A. Plumley/

David A. Plumley

Reg. No. 37,208

626/795-9900

DAP/jhg
Enclosures

Under the Paperwork Reduction Act of 1995, no persons are required to respond to a collection of information unless it contains a valid OMB control number.

Application Data Sheet 37 CFR 1.76		Attorney Docket Number	124598/412563-00010
		Application Number	15201196
Title of Invention	ACCELERATED ERASURE CODING SYSTEM AND METHOD		
<p>The application data sheet is part of the provisional or nonprovisional application for which it is being submitted. The following form contains the bibliographic data arranged in a format specified by the United States Patent and Trademark Office as outlined in 37 CFR 1.76. This document may be completed electronically and submitted to the Office in electronic format using the Electronic Filing System (EFS) or the document may be printed and included in a paper filed application.</p>			

Secrecy Order 37 CFR 5.2:

Portions or all of the application associated with this Application Data Sheet may fall under a Secrecy Order pursuant to 37 CFR 5.2 (Paper filers only. Applications that fall under Secrecy Order may not be filed electronically.)

Inventor Information:

Inventor 1 Remove				
Legal Name				
Prefix	Given Name	Middle Name	Family Name	Suffix
	Michael	H.	Anderson	
Residence Information (Select One) <input checked="" type="radio"/> US Residency <input type="radio"/> Non US Residency <input type="radio"/> Active US Military Service				
City	Los Angeles	State/Province	CA	Country of Residence
				US

Mailing Address of Inventor:

Address 1	6423 Monterey Road, Unit 2			
Address 2				
City	Los Angeles	State/Province	CA	
Postal Code	90042	Country	US	

Inventor 2 Remove				
Legal Name				
Prefix	Given Name	Middle Name	Family Name	Suffix
	Sarah		Mann	
Residence Information (Select One) <input checked="" type="radio"/> US Residency <input type="radio"/> Non US Residency <input type="radio"/> Active US Military Service				
City	Oakland	State/Province	CA	Country of Residence
				US

Mailing Address of Inventor:

Address 1	196 Ridgeway Avenue			
Address 2				
City	Oakland	State/Province	CA	
Postal Code	94611	Country	US	

All Inventors Must Be Listed - Additional inventor information blocks may be generated within this form by selecting the Add button.

Add

Correspondence Information:

Enter either Customer Number or complete the Correspondence Information section below. For further information see 37 CFR 1.33(a).

Under the Paperwork Reduction Act of 1995, no persons are required to respond to a collection of information unless it contains a valid OMB control number.

Application Data Sheet 37 CFR 1.76	Attorney Docket Number	124596/412563-00010
	Application Number	
Title of Invention	ACCELERATED ERASURE CODING SYSTEM AND METHOD	

 An Address is being provided for the correspondence information of this application.

Customer Number	23363		
Email Address	PTO@LRRC.COM	<input type="button" value="Add Email"/>	<input type="button" value="Remove Email"/>

Application Information:

Title of the Invention	ACCELERATED ERASURE CODING SYSTEM AND METHOD		
Attorney Docket Number	124596/412563-00010	Small Entity Status Claimed	<input checked="" type="checkbox"/>
Application Type	Nonprovisional		
Subject Matter	Utility		
Total Number of Drawing Sheets (if any)	9	Suggested Figure for Publication (if any)	

Filing By Reference:

Only complete this section when filing an application by reference under 35 U.S.C. 111(c) and 37 CFR 1.57(a). Do not complete this section if application papers including a specification and any drawings are being filed. Any domestic benefit or foreign priority information must be provided in the appropriate section(s) below (i.e., "Domestic Benefit/National Stage Information" and "Foreign Priority Information").

For the purposes of a filing date under 37 CFR 1.53(b), the description and any drawings of the present application are replaced by this reference to the previously filed application, subject to conditions and requirements of 37 CFR 1.57(a).

Application number of the previously filed application	Filing date (YYYY-MM-DD)	Intellectual Property Authority or Country

Publication Information:
 Request Early Publication (Fee required at time of Request 37 CFR 1.219)

Request Not to Publish. I hereby request that the attached application not be published under 35 U.S.C. 122(b) and certify that the invention disclosed in the attached application **has not and will not** be the subject of an application filed in another country, or under a multilateral international agreement, that requires publication at eighteen months after filing.

Representative Information:

Representative information should be provided for all practitioners having a power of attorney in the application. Providing this information in the Application Data Sheet does not constitute a power of attorney in the application (see 37 CFR 1.32). Either enter Customer Number or complete the Representative Name section below. If both sections are completed the customer number will be used for the Representative Information during processing.

Please Select One:	<input checked="" type="radio"/> Customer Number	<input type="radio"/> US Patent Practitioner	<input type="radio"/> Limited Recognition (37 CFR 11.9)
Customer Number	23363		

Under the Paperwork Reduction Act of 1995, no persons are required to respond to a collection of information unless it contains a valid OMB control number.

Application Data Sheet 37 CFR 1.76		Attorney Docket Number	124596/412563-00010
		Application Number	
Title of Invention	ACCELERATED ERASURE CODING SYSTEM AND METHOD		

Domestic Benefit/National Stage Information:

This section allows for the applicant to either claim benefit under 35 U.S.C. 119(e), 120, 121, 365(c), or 386(c) or indicate National Stage entry from a PCT application. Providing benefit claim information in the Application Data Sheet constitutes the specific reference required by 35 U.S.C. 119(e) or 120, and 37 CFR 1.78.

When referring to the current application, please leave the "Application Number" field blank.

Prior Application Status		Patented		Remove	
Application Number	Continuity Type	Prior Application Number	Filing Date (YYYY-MM-DD)	Patent Number	Issue Date (YYYY-MM-DD)
15201196	Continuation of	14852438	2015-09-11	9385759	2016-07-05
Prior Application Status		Patented		Remove	
Application Number	Continuity Type	Prior Application Number	Filing Date (YYYY-MM-DD)	Patent Number	Issue Date (YYYY-MM-DD)
14852438	Continuation of	14223740	2014-03-24	9160374	2015-10-13
Prior Application Status		Patented		Remove	
Application Number	Continuity Type	Prior Application Number	Filing Date (YYYY-MM-DD)	Patent Number	Issue Date (YYYY-MM-DD)
14223740	Continuation of	13341833	2011-12-30	8683296	2014-03-25

Additional Domestic Benefit/National Stage Data may be generated within this form by selecting the **Add** button.

Foreign Priority Information:

This section allows for the applicant to claim priority to a foreign application. Providing this information in the application data sheet constitutes the claim for priority as required by 35 U.S.C. 119(b) and 37 CFR 1.55. When priority is claimed to a foreign application that is eligible for retrieval under the priority document exchange program (PDX), the information will be used by the Office to automatically attempt retrieval pursuant to 37 CFR 1.55(i)(1) and (2). Under the PDX program, applicant bears the ultimate responsibility for ensuring that a copy of the foreign application is received by the Office from the participating foreign intellectual property office, or a certified copy of the foreign priority application is filed, within the time period specified in 37 CFR 1.55(g)(1).

Remove			
Application Number	Country ¹	Filing Date (YYYY-MM-DD)	Access Code ¹ (if applicable)

Additional Foreign Priority Data may be generated within this form by selecting the **Add** button.

Statement under 37 CFR 1.55 or 1.78 for AIA (First Inventor to File) Transition Applications

Under the Paperwork Reduction Act of 1995, no persons are required to respond to a collection of information unless it contains a valid OMB control number.

Application Data Sheet 37 CFR 1.76	Attorney Docket Number	124596/412563-00010
	Application Number	
Title of Invention	ACCELERATED ERASURE CODING SYSTEM AND METHOD	

<input type="checkbox"/> This application (1) claims priority to or the benefit of an application filed before March 16, 2013 and (2) also contains, or contained at any time, a claim to a claimed invention that has an effective filing date on or after March 16, 2013. NOTE: By providing this statement under 37 CFR 1.55 or 1.76, this application, with a filing date on or after March 16, 2013, will be examined under the first inventor to file provisions of the AIA.

Under the Paperwork Reduction Act of 1995, no persons are required to respond to a collection of information unless it contains a valid OMB control number.

Application Data Sheet 37 CFR 1.76	Attorney Docket Number	124596/412563-00010
	Application Number	
Title of invention	ACCELERATED ERASURE CODING SYSTEM AND METHOD	

Authorization or Opt-Out of Authorization to Permit Access:

When this Application Data Sheet is properly signed and filed with the application, applicant has provided written authority to permit a participating foreign intellectual property (IP) office access to the instant application-as-filed (see paragraph A in subsection 1 below) and the European Patent Office (EPO) access to any search results from the instant application (see paragraph B in subsection 1 below).

Should applicant choose not to provide an authorization identified in subsection 1 below, applicant **must opt-out** of the authorization by checking the corresponding box A or B or both in subsection 2 below.

NOTE: This section of the Application Data Sheet is **ONLY** reviewed and processed with the **INITIAL** filing of an application. After the initial filing of an application, an Application Data Sheet cannot be used to provide or rescind authorization for access by a foreign IP office(s). Instead, Form PTO/SB/39 or PTO/SB/69 must be used as appropriate.

1. Authorization to Permit Access by a Foreign Intellectual Property Office(s)

A. Priority Document Exchange (PDX) - Unless box A in subsection 2 (opt-out of authorization) is checked, the undersigned hereby **grants the USPTO authority** to provide the European Patent Office (EPO), the Japan Patent Office (JPO), the Korean Intellectual Property Office (KIPO), the State Intellectual Property Office of the People's Republic of China (SIPO), the World Intellectual Property Organization (WIPO), and any other foreign intellectual property office participating with the USPTO in a bilateral or multilateral priority document exchange agreement in which a foreign application claiming priority to the instant patent application is filed, access to: (1) the instant patent application-as-filed and its related bibliographic data, (2) any foreign or domestic application to which priority or benefit is claimed by the instant application and its related bibliographic data, and (3) the date of filing of this Authorization. See 37 CFR 1.14(h)(1).

B. Search Results from U.S. Application to EPO - Unless box B in subsection 2 (opt-out of authorization) is checked, the undersigned hereby **grants the USPTO authority** to provide the EPO access to the bibliographic data and search results from the instant patent application when a European patent application claiming priority to the instant patent application is filed. See 37 CFR 1.14(h)(2).

The applicant is reminded that the EPO's Rule 141(1) EPC (European Patent Convention) requires applicants to submit a copy of search results from the instant application without delay in a European patent application that claims priority to the instant application.

2. Opt-Out of Authorizations to Permit Access by a Foreign Intellectual Property Office(s)

A. Applicant **DOES NOT** authorize the USPTO to permit a participating foreign IP office access to the instant application-as-filed. If this box is checked, the USPTO will not be providing a participating foreign IP office with any documents and information identified in subsection 1A above.

B. Applicant **DOES NOT** authorize the USPTO to transmit to the EPO any search results from the instant patent application. If this box is checked, the USPTO will not be providing the EPO with search results from the instant application.

NOTE: Once the application has published or is otherwise publicly available, the USPTO may provide access to the application in accordance with 37 CFR 1.14.

Under the Paperwork Reduction Act of 1995, no persons are required to respond to a collection of information unless it contains a valid OMB control number.

Application Data Sheet 37 CFR 1.76	Attorney Docket Number	124596/412563-00010
	Application Number	
Title of Invention	ACCELERATED ERASURE CODING SYSTEM AND METHOD	

Applicant Information:

Providing assignment information in this section does not substitute for compliance with any requirement of part 3 of Title 37 of CFR to have an assignment recorded by the Office

Applicant 1

If the applicant is the inventor (or the remaining joint inventor or inventors under 37 CFR 1.45), this section should not be completed. The information to be provided in this section is the name and address of the legal representative who is the applicant under 37 CFR 1.43; or the name and address of the assignee, person to whom the inventor is under an obligation to assign the invention, or person who otherwise shows sufficient proprietary interest in the matter who is the applicant under 37 CFR 1.46. If the applicant is an applicant under 37 CFR 1.46 (assignee, person to whom the inventor is obligated to assign, or person who otherwise shows sufficient proprietary interest) together with one or more joint inventors, then the joint inventor or inventors who are also the applicant should be identified in this section.

Assignee
 Legal Representative under 35 U.S.C. 117
 Joint Inventor
 Person to whom the inventor is obligated to assign.
 Person who shows sufficient proprietary interest

If applicant is the legal representative, indicate the authority to file the patent application, the inventor is:

Name of the Deceased or Legally Incapacitated Inventor:

If the Applicant is an Organization check here.

Organization Name:

Mailing Address Information For Applicant:

Address 1	8425 Monterey Road, Unit 2 7515 Bosque Blvd., Suite 203		
Address 2			
City	Los Angeles <u>Waco</u>	State/Province	CA <u>TX</u>
Country	US	Postal Code	90042 <u>76710</u>
Phone Number		Fax Number	
Email Address			

Additional Applicant Data may be generated within this form by selecting the Add button.

Assignee Information including Non-Applicant Assignee Information:

Providing assignment information in this section does not substitute for compliance with any requirement of part 3 of Title 37 of CFR to have an assignment recorded by the Office.

Under the Paperwork Reduction Act of 1995, no persons are required to respond to a collection of information unless it contains a valid OMB control number.

Application Data Sheet 37 CFR 1.76	Attorney Docket Number	124596/412563-00010
	Application Number	
Title of Invention	ACCELERATED ERASURE CODING SYSTEM AND METHOD	

Assignee 1

Complete this section if assignee information, including non-applicant assignee information, is desired to be included on the patent application publication. An assignee-applicant identified in the "Applicant Information" section will appear on the patent application publication as an applicant. For an assignee-applicant, complete this section only if identification as an assignee is also desired on the patent application publication.

If the Assignee or Non-Applicant Assignee is an Organization check here.

Prefix	Given Name	Middle Name	Family Name	Suffix

Mailing Address Information For Assignee including Non-Applicant Assignee:

Address 1				
Address 2				
City		State/Province		
Country ¹	Postal Code			
Phone Number		Fax Number		
Email Address				

Additional Assignee or Non-Applicant Assignee Data may be generated within this form by selecting the Add button.

Signature:

NOTE: This Application Data Sheet must be signed in accordance with 37 CFR 1.33(b). However, if this Application Data Sheet is submitted with the **INITIAL** filing of the application and either box A or B is not checked in subsection 2 of the "Authorization or Opt-Out of Authorization to Permit Access" section, then this form must also be signed in accordance with 37 CFR 1.14(c).

This Application Data Sheet **must** be signed by a patent practitioner if one or more of the applicants is a **juristic entity** (e.g., corporation or association). If the applicant is two or more joint inventors, this form must be signed by a patent practitioner, **all** joint inventors who are the applicant, or one or more joint inventor-applicants who have been given power of attorney (e.g., see USPTO Form PTO/AIA/81) on behalf of **all** joint inventor-applicants.

See 37 CFR 1.4(d) for the manner of making signatures and certifications.

Signature	/David A. Plumley/		Date (YYYY-MM-DD)	2021-02-23	
First Name	David A.	Last Name	Plumley	Registration Number	37208

Additional Signature may be generated within this form by selecting the Add button.

Under the Paperwork Reduction Act of 1995, no persons are required to respond to a collection of information unless it contains a valid OMB control number.

Application Data Sheet 37 CFR 1.76	Attorney Docket Number	124596/412563-00010
	Application Number	
Title of Invention	ACCELERATED ERASURE CODING SYSTEM AND METHOD	

This collection of information is required by 37 CFR 1.76. The information is required to obtain or retain a benefit by the public which is to file (and by the USPTO to process) an application. Confidentiality is governed by 35 U.S.C. 122 and 37 CFR 1.14. This collection is estimated to take 23 minutes to complete, including gathering, preparing, and submitting the completed application data sheet form to the USPTO. Time will vary depending upon the individual case. Any comments on the amount of time you require to complete this form and/or suggestions for reducing this burden, should be sent to the Chief Information Officer, U.S. Patent and Trademark Office, U.S. Department of Commerce, P.O. Box 1450, Alexandria, VA 22313-1450. **DO NOT SEND FEES OR COMPLETED FORMS TO THIS ADDRESS. SEND TO: Commissioner for Patents, P.O. Box 1450, Alexandria, VA 22313-1450.**

Privacy Act Statement

The Privacy Act of 1974 (P.L. 93-579) requires that you be given certain information in connection with your submission of the attached form related to a patent application or patent. Accordingly, pursuant to the requirements of the Act, please be advised that: (1) the general authority for the collection of this information is 35 U.S.C. 2(b)(2); (2) furnishing of the information solicited is voluntary; and (3) the principal purpose for which the information is used by the U.S. Patent and Trademark Office is to process and/or examine your submission related to a patent application or patent. If you do not furnish the requested information, the U.S. Patent and Trademark Office may not be able to process and/or examine your submission, which may result in termination of proceedings or abandonment of the application or expiration of the patent.

The information provided by you in this form will be subject to the following routine uses:

1. The information on this form will be treated confidentially to the extent allowed under the Freedom of Information Act (5 U.S.C. 552) and the Privacy Act (5 U.S.C. 552a). Records from this system of records may be disclosed to the Department of Justice to determine whether the Freedom of Information Act requires disclosure of these records.
2. A record from this system of records may be disclosed, as a routine use, in the course of presenting evidence to a court, magistrate, or administrative tribunal, including disclosures to opposing counsel in the course of settlement negotiations.
3. A record in this system of records may be disclosed, as a routine use, to a Member of Congress submitting a request involving an individual, to whom the record pertains, when the individual has requested assistance from the Member with respect to the subject matter of the record.
4. A record in this system of records may be disclosed, as a routine use, to a contractor of the Agency having need for the information in order to perform a contract. Recipients of information shall be required to comply with the requirements of the Privacy Act of 1974, as amended, pursuant to 5 U.S.C. 552a(m).
5. A record related to an International Application filed under the Patent Cooperation Treaty in this system of records may be disclosed, as a routine use, to the International Bureau of the World Intellectual Property Organization, pursuant to the Patent Cooperation Treaty.
6. A record in this system of records may be disclosed, as a routine use, to another federal agency for purposes of National Security review (35 U.S.C. 181) and for review pursuant to the Atomic Energy Act (42 U.S.C. 218(c)).
7. A record from this system of records may be disclosed, as a routine use, to the Administrator, General Services, or his/her designee, during an inspection of records conducted by GSA as part of that agency's responsibility to recommend improvements in records management practices and programs, under authority of 44 U.S.C. 2904 and 2906. Such disclosure shall be made in accordance with the GSA regulations governing inspection of records for this purpose, and any other relevant (i.e., GSA or Commerce) directive. Such disclosure shall not be used to make determinations about individuals.
8. A record from this system of records may be disclosed, as a routine use, to the public after either publication of the application pursuant to 35 U.S.C. 122(b) or issuance of a patent pursuant to 35 U.S.C. 151. Further, a record may be disclosed, subject to the limitations of 37 CFR 1.14, as a routine use, to the public if the record was filed in an application which became abandoned or in which the proceedings were terminated and which application is referenced by either a published application, an application open to public inspections or an issued patent.
9. A record from this system of records may be disclosed, as a routine use, to a Federal, State, or local law enforcement agency, if the USPTO becomes aware of a violation or potential violation of law or regulation.

PATENT

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

Inventor(s)	: Michael H. Anderson et al.	Confirmation No. 1895
Assignee	: STREAMSCALE, INC.	
Patent No.	: 10,003,358	
Issued	: June 19, 2018	
Application No.	: 15/201,196	
Filed	: July 1, 2016	
Title	: ACCELERATED ERASURE CODING SYSTEM AND METHOD	
Docket No.	: 124596 (411563-00010)	

STATEMENT OF MICHAEL H. ANDERSON IN SUPPORT OF PETITION FOR CORRECTION OF INVENTORSHIP PURSUANT TO 37 C.F.R. § 1.324

Commissioner for Patents
P.O. Box 1450
Alexandria, VA 22313-1450

Post Office Box 29001
Glendale, CA 91209-9001

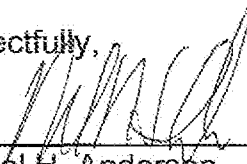
Commissioner:

I, the undersigned, declare and state as follows:

I submit this statement in support of STREAMSCALE, INC.'s petition to correct the inventorship in the above-identified patent. I am the named inventor of the above-identified patent. I understand that the petition seeks to add SARAH MANN as an inventor to this patent and I agree to the requested change of inventorship.

Executed this 20 of Feb, 2021 in Udon Thani,
Thailand

Respectfully,



Michael H. Anderson

DAP/srd

PATENT

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

Inventor(s)	: Michael H. Anderson et al.	Confirmation No. 1895
Assignee	: STREAMSCALE, INC.	
Patent No.	: 10,003,358	
Issued	: June 19, 2018	
Application No.	: 15/201,196	
Filed	: July 1, 2016	
Title	: ACCELERATED ERASURE CODING SYSTEM AND METHOD	
Docket No.	: 124596 (411563-00010)	

**STATEMENT OF ASSIGNEE IN SUPPORT OF PETITION
FOR CORRECTION OF INVENTORSHIP UNDER 37 C.F.R. § 1.324 AND
COMPLYING WITH 37 C.F.R. § 3.73(c)**

Commissioner for Patents
P.O. Box 1450
Alexandria, VA 22313-1450

Post Office Box 29001
Glendale, CA 91209-9001

The below signed individual declares as follows:

1. I am authorized to act on behalf of STREAMSCALE, INC. and have the title indicated below.

2. STREAMSCALE, INC. is the assignee of the entire interest of the patent identified above, by virtue of the following Assignments from the inventors.


(a) An Assignment of this invention by inventor Michael H. Anderson was recorded on February 28, 2018 at Reel No. 045061 and Frame No. 0217.

(b) A second Assignment of this invention by inventor Sarah Mann, the inventor to be added on this patent, is attached hereto.

3. The Assignee agrees to the addition of Sarah Mann as an inventor on the patent.

Date Feb 20, 2021

STREAMSCALE, INC.

By: 
Name: Michael H. Anderson
Title: President

DAP/srd

**INVENTOR'S DECLARATION AND ASSIGNMENT
FOR PATENT APPLICATION**

PATENT

Title of Invention: ACCELERATED ERASURE CODING SYSTEM AND METHOD

Docket No.: 124596 (411563-00010)

Application No. 15/201,196

INVENTOR'S DECLARATION

As a below named inventor, I hereby declare that:

This declaration is directed to the attached application unless the following is checked:

United States Application or PCT International Application Number 15/201,196 filed on July 1, 2016.

The above-identified application was made or authorized to be made by me.

I believe that I am the original inventor or an original joint inventor of a claimed invention in the above-identified application.

I have reviewed and understand the contents of the above-identified application, including the claims.

I acknowledge the duty to disclose information which is material to patentability as defined in 37 C.F.R. § 1.56, including for continuation-in-part applications, material information which became available between the filing date of the prior application and the national or PCT international filing date of the continuation-in-part application.

I acknowledge that any willful false statement made in this declaration is punishable under 18 U.S.C. § 1001 by fine or imprisonment of not more than five (5) years, or both.

ASSIGNMENT

In consideration of good and valuable consideration, the receipt of which is hereby acknowledged, the undersigned,

(1) Sarah Mann

HEREBY SELL(S), ASSIGN(S) AND TRANSFER(S) TO

(2) STREAMSCALE, INC.

having a place of business at

(3) 7215 Bosque Blvd., Suite 203, Waco, Texas 76710

(hereinafter called "ASSIGNEE") the entire right, title and interest in and to any and all improvements which are disclosed in the application for United States Letters Patent entitled

(4) ACCELERATED ERASURE CODING SYSTEM AND METHOD

which application was executed on even date herewith or was

**INVENTOR'S DECLARATION AND ASSIGNMENT
FOR PATENT APPLICATION**

Docket No.: 124596 (411563-00010)
Application No.: 15/201,196

(a) executed on (5a): _____;
(b) filed on (5b): July 1, 2016 _____,
Application No.: 15/201,196 _____;

(LEWIS ROCA ROTHGERBER CHRISTIE
LLP, P.O. Box 29001, Glendale, CA 91209-
9001) is hereby authorized to insert in (b) the
specified data, when known.

including any and all United States Patents
which may be granted on said application, and any and all extensions, divisions, reissues,
substitutes, renewals or continuations of said application and patents, and the right to all benefits
under all international conventions for the protection of industrial property and applications for
said improvements.

It is hereby authorized and requested that the Commissioner of Patents issue any and all of said
Letters Patent, when granted, to said ASSIGNEE, its assigns or its successors in interest or its
designee.

Upon said consideration, it is further agreed that, when requested, without charge to but at the
expense of said ASSIGNEE, the undersigned will execute all divisional, continuing, substitute,
renewal, and reissue patent applications; execute all rightful other papers; and generally do
everything possible which said ASSIGNEE shall consider desirable for aiding in securing and
maintaining patent protection as provided herein.

Sarah Mann
Legal Name of Inventor

2/18/2021
Date

DocuSigned by:
/Sarah Mann/
Signature

WITNESSES:

PATENT

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

Inventor(s) : Michael H. Anderson et al. Confirmation No. 1895
Assignee : STREAMSCALE, INC.
Patent No. : 10,003,358
Issued : June 19, 2018
Application No. : 15/201,196
Filed : July 1, 2016
Title : ACCELERATED ERASURE CODING SYSTEM AND METHOD
Docket No. : 124596 (411563-00010)

STATEMENT OF SARAH MANN IN SUPPORT OF PETITION FOR CORRECTION OF INVENTORSHIP PURSUANT TO 37 C.F.R. § 1.324

Commissioner for Patents
P.O. Box 1450
Alexandria, VA 22313-1450

Post Office Box 29001
Glendale, CA 91209-9001


Commissioner:

I, the undersigned, declare and state as follows:

I submit this statement in support of STREAMSCALE, INC.'s petition to correct the inventorship in the above-identified patent. I understand that the petition seeks to add me, the undersigned, as an inventor to this patent and I agree to the requested change of inventorship.

Executed this 18 of February, 2021 in Oakland,
CA

Respectfully,

DocuSigned by:

9574763A724343D
Sarah Mann

DAP/srd

**INVENTOR'S DECLARATION AND ASSIGNMENT
FOR PATENT APPLICATION**

PATENT

Title of Invention: ACCELERATED ERASURE CODING SYSTEM AND METHOD

Docket No.: 124596 (411563-00010)

Application No. 15/201,196

INVENTOR'S DECLARATION

As a below named inventor, I hereby declare that:

This declaration is directed to the attached application unless the following is checked:

United States Application or PCT International Application Number 15/201,196 filed on July 1, 2016.

The above-identified application was made or authorized to be made by me.

I believe that I am the original inventor or an original joint inventor of a claimed invention in the above-identified application.

I have reviewed and understand the contents of the above-identified application, including the claims.

I acknowledge the duty to disclose information which is material to patentability as defined in 37 C.F.R. § 1.56, including for continuation-in-part applications, material information which became available between the filing date of the prior application and the national or PCT international filing date of the continuation-in-part application.

I acknowledge that any willful false statement made in this declaration is punishable under 18 U.S.C. § 1001 by fine or imprisonment of not more than five (5) years, or both.

ASSIGNMENT

In consideration of good and valuable consideration, the receipt of which is hereby acknowledged, the undersigned,

(1) Sarah Mann

HEREBY SELL(S), ASSIGN(S) AND TRANSFER(S) TO

(2) STREAMSCALE, INC.

having a place of business at

(3) 7215 Bosque Blvd., Suite 203, Waco, Texas 76710

(hereinafter called "ASSIGNEE") the entire right, title and interest in and to any and all improvements which are disclosed in the application for United States Letters Patent entitled

(4) ACCELERATED ERASURE CODING SYSTEM AND METHOD

which application was executed on even date herewith or was

**INVENTOR'S DECLARATION AND ASSIGNMENT
FOR PATENT APPLICATION**

Docket No.: 124596 (411563-00010)
Application No.: 15/201,196

(a) executed on (5a): _____;
(b) filed on (5b): July 1, 2016 _____,
Application No.: 15/201,196 _____;

(LEWIS ROCA ROTHGERBER CHRISTIE
LLP, P.O. Box 29001, Glendale, CA 91209-
9001) is hereby authorized to insert in (b) the
specified data, when known.

including any and all United States Patents
which may be granted on said application, and any and all extensions, divisions, reissues,
substitutes, renewals or continuations of said application and patents, and the right to all benefits
under all international conventions for the protection of industrial property and applications for
said improvements.

It is hereby authorized and requested that the Commissioner of Patents issue any and all of said
Letters Patent, when granted, to said ASSIGNEE, its assigns or its successors in interest or its
designee.

Upon said consideration, it is further agreed that, when requested, without charge to but at the
expense of said ASSIGNEE, the undersigned will execute all divisional, continuing, substitute,
renewal, and reissue patent applications; execute all rightful other papers; and generally do
everything possible which said ASSIGNEE shall consider desirable for aiding in securing and
maintaining patent protection as provided herein.

Sarah Mann
Legal Name of Inventor

2/18/2021
Date

DocuSigned by:
/Sarah Mann/
Signature

WITNESSES:

Electronic Patent Application Fee Transmittal

Application Number:	15201196			
Filing Date:	01-Jul-2016			
Title of Invention:	ACCELERATED ERASURE CODING SYSTEM AND METHOD			
First Named Inventor/Applicant Name:	Michael H. Anderson			
Filer:	David A. Plumley/Jennifer Guerra			
Attorney Docket Number:	124596/411563-00010			
Filed as Small Entity				
Filing Fees for Utility under 35 USC 111(a)				
Description	Fee Code	Quantity	Amount	Sub-Total in USD(\$)
Basic Filing:				
Pages:				
Claims:				
Miscellaneous-Filing:				
Petition:				
Patent-Appeals-and-Interference:				
Post-Allowance-and-Post-Issuance:				
PROCESSING FEE CORRECTING INVENTORSHIP	2816	1	160	160

Description	Fee Code	Quantity	Amount	Sub-Total in USD(\$)
Extension-of-Time:				
Miscellaneous:				
Total in USD (\$)				160

Electronic Acknowledgement Receipt

EFS ID:	42002759
Application Number:	15201196
International Application Number:	
Confirmation Number:	1895
Title of Invention:	ACCELERATED ERASURE CODING SYSTEM AND METHOD
First Named Inventor/Applicant Name:	Michael H. Anderson
Customer Number:	23363
Filer:	David A. Plumley/Jennifer Guerra
Filer Authorized By:	David A. Plumley
Attorney Docket Number:	124596/411563-00010
Receipt Date:	23-FEB-2021
Filing Date:	01-JUL-2016
Time Stamp:	20:16:14
Application Type:	Utility under 35 USC 111(a)

Payment information:

Submitted with Payment	yes
Payment Type	CARD
Payment was successfully received in RAM	\$160
RAM confirmation Number	E20212MK16314302
Deposit Account	
Authorized User	

The Director of the USPTO is hereby authorized to charge indicated fees and credit any overpayment as follows:

File Listing:

Document Number	Document Description	File Name	File Size(Bytes)/ Message Digest	Multi Part /.zip	Pages (if appl.)
1	Petition for review by the Office of Petitions	124596_Petition.pdf	105472	no	2
			f7eba6fd26cc6344e63dea3d44b81f3b7c96bfed		
Warnings:					
Information:					
2	Application Data Sheet	124596_CorrectedADS.pdf	4775871	no	9
			5e5cd8ba12bc73ec9577568210cac6c6dbca6e3aa		
Warnings:					
Information:					
This is not an USPTO supplied ADS fillable form					
3	Examination support document	124596_Anderson_Stm.pdf	294382	no	1
			b22856785f3fc0c67cf0dabe30c1cb979b55f9a0		
Warnings:					
Information:					
4	Examination support document	124596_StreamScale_Stm.pdf	452860	no	3
			1684c07553306143d3c05df3319dad9820445ee1		
Warnings:					
Information:					
5	Examination support document	124596_Mann_Stm.pdf	206157	no	1
			476d14ce12d287c13498f7f069e5ecdca6f5e50b		
Warnings:					
The PDF file has been signed with a digital signature and the legal effect of the document will be based on the contents of the file not the digital signature.					
Information:					
6	Oath or Declaration filed	124596_Mann_DeclAsg.pdf	211412	no	2
			02169f30b7c2691b2c394bdb2a28dd4ef4a3ba39		

Warnings:

The PDF file has been signed with a digital signature and the legal effect of the document will be based on the contents of the file not the digital signature.

Information:

7	Fee Worksheet (SB06)	fee-info.pdf	30467	no	2
			7d9072e4b519ad1630619f8a6b747a7b78069fcd		

Warnings:

Information:

Total Files Size (in bytes):	6076621
-------------------------------------	---------

This Acknowledgement Receipt evidences receipt on the noted date by the USPTO of the indicated documents, characterized by the applicant, and including page counts, where applicable. It serves as evidence of receipt similar to a Post Card, as described in MPEP 503.

New Applications Under 35 U.S.C. 111

If a new application is being filed and the application includes the necessary components for a filing date (see 37 CFR 1.53(b)-(d) and MPEP 506), a Filing Receipt (37 CFR 1.54) will be issued in due course and the date shown on this Acknowledgement Receipt will establish the filing date of the application.

National Stage of an International Application under 35 U.S.C. 371

If a timely submission to enter the national stage of an international application is compliant with the conditions of 35 U.S.C. 371 and other applicable requirements a Form PCT/DO/EO/903 indicating acceptance of the application as a national stage submission under 35 U.S.C. 371 will be issued in addition to the Filing Receipt, in due course.

New International Application Filed with the USPTO as a Receiving Office

If a new international application is being filed and the international application includes the necessary components for an international filing date (see PCT Article 11 and MPEP 1810), a Notification of the International Application Number and of the International Filing Date (Form PCT/RO/105) will be issued in due course, subject to prescriptions concerning national security, and the date shown on this Acknowledgement Receipt will establish the international filing date of the application.

EXHIBIT F



US010291259B2

(12) **United States Patent**
Anderson

(10) **Patent No.:** **US 10,291,259 B2**
(45) **Date of Patent:** **May 14, 2019**

(54) **ACCELERATED ERASURE CODING SYSTEM AND METHOD**

(71) Applicant: **STREAMSCALE, INC.**, Los Angeles, CA (US)

(72) Inventor: **Michael H. Anderson**, Los Angeles, CA (US)

(73) Assignee: **STREAMSCALE, INC.**, Los Angeles, CA (US)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 0 days.

(21) Appl. No.: **15/976,175**

(22) Filed: **May 10, 2018**

(65) **Prior Publication Data**
US 2018/0262212 A1 Sep. 13, 2018

Related U.S. Application Data
(63) Continuation of application No. 15/201,196, filed on Jul. 1, 2016, now Pat. No. 10,003,358, which is a (Continued)

(51) **Int. Cl.**
H03M 13/15 (2006.01)
G06F 11/10 (2006.01)
(Continued)

(52) **U.S. Cl.**
CPC **H03M 13/154** (2013.01); **G06F 11/1068** (2013.01); **G06F 11/1076** (2013.01);
(Continued)

(58) **Field of Classification Search**
CPC H03M 13/154; H03M 13/1191; H03M 13/134; H03M 13/151; H03M 13/373;
(Continued)

(56) **References Cited**

U.S. PATENT DOCUMENTS

5,577,054 A 11/1996 Pharris
5,754,563 A 5/1998 White
(Continued)

OTHER PUBLICATIONS

Casey Henderson: Letter to the USENIX Community <https://www.usenix.org/system/files/conference/fast13/fast13_memo_021715.pdf>Feb. 17,2015.

(Continued)

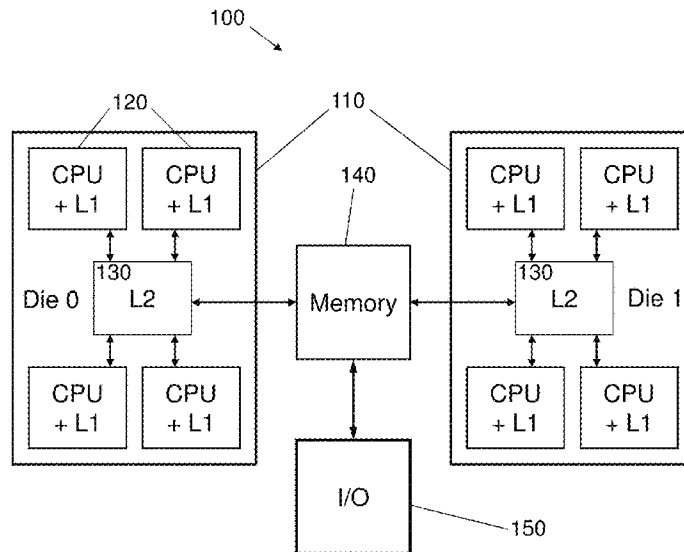
Primary Examiner — John J Tabone, Jr.

(74) *Attorney, Agent, or Firm* — Lewis Roca Rothgerber Christie LLP

(57) **ABSTRACT**

An accelerated erasure coding system includes a processing core for executing computer instructions and accessing data from a main memory, and a non-volatile storage medium for storing the computer instructions. The processing core, storage medium, and computer instructions are configured to implement an erasure coding system, which includes: a data matrix for holding original data in the main memory; a check matrix for holding check data in the main memory; an encoding matrix for holding first factors in the main memory, the first factors being for encoding the original data into the check data; and a thread for executing on the processing core. The thread includes: a parallel multiplier for concurrently multiplying multiple entries of the data matrix by a single entry of the encoding matrix; and a first sequencer for ordering operations through the data matrix and the encoding matrix using the parallel multiplier to generate the check data.

57 Claims, 9 Drawing Sheets



US 10,291,259 B2

Page 2

Related U.S. Application Data

continuation of application No. 14/852,438, filed on Sep. 11, 2015, now Pat. No. 9,385,759, which is a continuation of application No. 14/223,740, filed on Mar. 24, 2014, now Pat. No. 9,160,374, which is a continuation of application No. 13/341,833, filed on Dec. 30, 2011, now Pat. No. 8,683,296.

(51) Int. Cl.

H03M 13/11 (2006.01)
H03M 13/13 (2006.01)
G06F 12/02 (2006.01)
G06F 12/06 (2006.01)
H03M 13/37 (2006.01)
H03M 13/00 (2006.01)
H04L 1/00 (2006.01)
G11C 29/52 (2006.01)

(52) U.S. Cl.

CPC **G06F 11/1092** (2013.01); **G06F 11/1096** (2013.01); **G06F 12/0238** (2013.01); **G06F 12/06** (2013.01); **G11C 29/52** (2013.01); **H03M 13/1191** (2013.01); **H03M 13/134** (2013.01); **H03M 13/1515** (2013.01); **H03M 13/373** (2013.01); **H03M 13/3761** (2013.01); **H03M 13/3776** (2013.01); **H03M 13/616** (2013.01); **H03M 13/6502** (2013.01); **H04L 1/0043** (2013.01); **H04L 1/0057** (2013.01); **G06F 2211/109** (2013.01); **G06F 2211/1057** (2013.01)

(58) Field of Classification Search

CPC H03M 13/3761; H03M 13/3776; H03M 13/616; H03M 13/6502; G06F 11/1068; G06F 11/1076; G06F 11/1092; G06F 11/1096; G06F 12/0238; G06F 12/06; G06F 2211/1057; G06F 2211/109; G11C 29/52; H04L 1/0043; H04L 1/0057
 USPC 714/6.24, 6.1, 6.11, 6.2, 6.21, 6.32, 763, 714/764, 752, 758, 768, 770, 773, 784, 714/786

See application file for complete search history.

(56)

References Cited

U.S. PATENT DOCUMENTS

6,486,803 B1 11/2002 Luby et al.
 6,654,924 B1* 11/2003 Hassner G11B 20/1813
 714/758
 6,823,425 B2* 11/2004 Ghosh G06F 11/1076
 711/114
 7,350,126 B2* 3/2008 Winograd G06F 11/1076
 714/752
 7,865,809 B1 1/2011 Lee et al.
 7,930,337 B2 4/2011 Hasenplaugh et al.
 8,145,941 B2* 3/2012 Jacobson G06F 11/1076
 714/6.24
 8,352,847 B2* 1/2013 Gunnam G06F 17/16
 714/758
 8,683,296 B2* 3/2014 Anderson H03M 13/1515
 714/763
 8,914,706 B2* 12/2014 Anderson G06F 11/1076
 714/6.24
 9,160,374 B2* 10/2015 Anderson H03M 13/3761
 9,258,014 B2* 2/2016 Anderson G06F 11/1076
 9,385,759 B2* 7/2016 Anderson H03M 13/3761
 10,003,358 B2* 6/2018 Anderson H03M 13/154
 2009/0055717 A1 2/2009 Au et al.
 2009/0249170 A1 10/2009 Maiuzzo
 2010/0293439 A1 11/2010 Flynn et al.

2011/0029756 A1* 2/2011 Biscondi H03M 13/1114
 712/22
 2012/0272036 A1* 10/2012 Muralimanohar .. G06F 12/0238
 711/202
 2013/0108048 A1* 5/2013 Grube H04W 12/00
 380/270
 2013/0110962 A1* 5/2013 Grube H04W 12/00
 709/213
 2013/0111552 A1* 5/2013 Grube H04W 12/00
 726/3
 2013/0124932 A1* 5/2013 Schuh G06F 9/44
 714/718
 2013/0173956 A1* 7/2013 Anderson G06F 11/1076
 714/6.24
 2013/0173996 A1* 7/2013 Anderson H03M 13/3761
 714/770
 2014/0040708 A1 2/2014 Maiuzzo
 2014/0068391 A1 3/2014 Goel et al.
 2015/0012796 A1* 1/2015 Anderson H03M 13/3761
 714/763
 2017/0005671 A1* 1/2017 Anderson H03M 13/3761

OTHER PUBLICATIONS

Chandan Kumar Singh: EC Jerasure plugin and StreamScale Inc, <<http://www.spinics.net/lists/ceph-devel/msg29944.html>> Apr. 20, 2016.
 Code Poetry and Text Adventures: <<http://catid.mechafetus.com/news/news.php?view=381>> Dec. 14, 2014.
 Curtis Chan: StreamScale Announces Settlement of Erasure Code Technology Patent Litigation, <<http://www.prweb.com/releases/2014/12/prweb12368357.htm>>, Dec. 3, 2014.
 Ethan Miller, <<https://plus.google.com/113956021908222328905/posts/bPcYevPkJWd>>, Aug. 2, 2013.
 H. Peter Anvin. "The mathematics of RAID-6." 2004, 2011.
 Hafner et al., Matrix Methods for Lost Data Reconstruction in Erasure Codes, Nov. 16, 2005, USenix Fast '05 Paper, pp. 1-26.
 James S. Plank, Ethan L. Miller, Will B. Houston: Ge-Complete: a Comprehensive Open Source Library for Galois Field Arithmetic, <<http://web.eecs.utk.edu/~plank/qplank/papers/Cs-13-703.html>> Jan. 2013.
 James S. Plank, Jianqiang Luo, Catherine D. Schuman, Lihao Xu, Zooko Wilcox-O'Hearn: A Performance Evaluation and Examination of Open-Source Erasure Coding Libraries for Storage, <<https://www.usenix.org/legacy/event/fast09/tech/fullpapers/plank/plank.html>> 2009.
 Kevin M. Greenan, Ethan L. Miller, Thomas J.E. Schwarz, S. J.: Optimizing Galois Field Arithmetic for Diverse Processor Architectures and Applications, *Proceedings of the 16th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (Mascots 2008)*, Baltimore, MD, Sep. 2008.
 Lee, "High-Speed VLSI Architecture for Parallel Reed-Solomon Decoder", IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 11, No. 2, Apr. 2003, pp. 288-294.
 Li et al.; Preventing Silent Data Corruptions from Propagating During Data Reconstruction; IEEE Transactions on Computers, vol. 59, No. 12, Dec. 2010; pp. 1611-1624.
 Li Han and Qian Huan-yan. "Parallelized Network Coding with SIMD instruction sets." in Computer Science and Computational Technology, 2008. ISCSCT'08. *International Symposium on*, vol. 1, pp. 364-369. IEEE, 2008.
 Loic Dachary: Deadline of Github pull request for Hammer release, <<http://www.spinics.net/lists/ceph-devel/msg22001.html>> Jan. 13, 2015.
 Louis Lavile: <<https://twitter.com/louislavile>> Nov. 13, 2014.
 M. Lalam, et al. "Sliding Encoding-Window for Reed-Solomon code decoding," 4th International Symposium on Turbo Codes & Related Topics; 6th International ITG-Conference on Source and Channel Coding, Munich, Germany, 2006, pp. 1-6.
 Maddock, et al.; White Paper, Surviving Two Disk Failures Introducing Various "RAID 6" Implementations; Xyratex; pp. 1-13.

US 10,291,259 B2

Page 3

(56)

References Cited

OTHER PUBLICATIONS

Mann, "The Original View of Reed-Solomon Coding and the Welch-Berlekamp Decoding Algorithm", A Dissertation Submitted to the Faculty of the Graduate Interdisciplinary Program in Applied Mathematics, The University of Arizona, Jul. 19, 2013, 143 sheets.

Marius Gedminas: <<http://eavesdrop.openstack.org/irclogs/%23openstack-swift/%23openstack-swift.2015-04-30.log.html>> Apr. 29, 2015.

Matthew L. Curry, Anthony Skjellum, H. Lee Ward, and Ron Brightwell. "Arbitrary dimension reed-solomon coding and decoding for extended raid on gpus." In Petascale Data Storage Workshop, 2008. PDSW'08. 3rd, pp. 1-3. IEEE, 2008.

Matthew L. Curry, Anthony Skjellum, H. Lee Ward, Ron Brightwell: Gibraltar: a Reed-Solomon coding library for storage applications on programmable graphics processors. *Concurrency and Computation: Practice and Experience* 23(18): pp. 2477-2495 (2011).

Matthew L. Curry, H. Lee Ward, Anthony Skjellum, Ron Brightwell: a Lightweight, Gpu-Based Software Raid System. *ICPP 2010*: pp. 565-572.

Matthew L. Curry, Lee H. Ward, Anthony Skjellum, and Ron B. Brightwell: Accelerating Reed-Solomon Coding in Raid Systems With GPUs, *Parallel and Distributed Processing, 2008. Ipdps 2008. IEEE International Symposium on*. IEEE, 2008.

Michael a. O'Shea: StreamScale, <<https://lists.ubuntu.com/archives/technical-board/2015-April/002100.html>> Apr. 29, 2015.

Mike Masnik: Patent Troll Kills Open Source Project on Speeding Up the Computation of Erasure Codes, <<https://www.techdirt.com/articles/20141115/07113529155/patent-troll-kills-open-source-project-speeding-up-computation-erasure-codes.shtml>>, Nov. 19, 2014.

Neifeld, M.A & Sridharan, S. K. (1997). Parallel error correction using spectral Reed-Solomon codes. *Journal of Optical Communications*, 18(4), pp. 144-150.

Plank; All About Erasure Codes: - Reed-Solomon Coding - LDPC Coding; Logistical Computing and Internetworking Laboratory, Department of Computer Science, University of Tennessee; ICL - Aug. 20, 2004; 52 sheets.

Robert Louis Cloud, Matthew L. Curry, H. Lee Ward, Anthony Skjellum, Purushotham Bangalore: Accelerating Lossless Data Compression with GPUs. CoRR abs/1107.1525 (2011).

Roy Schestowitz: US Patent Reform (on Trolls Only) More or Less Buried or Ineffective, <<http://techrighs.org/2014/12/12/us-patent-reform/>> Dec. 12, 2014.

Weibin Sun, Robert Ricci, Matthew L. Curry: GPUstore: harnessing Gpu computing for storage systems in the OS kernel. *SYSTOR 2012*: p. 6.

Xin Zhou, Anthony Skjellum, Matthew L. Curry: Abstract: Extended Abstract for Evaluating Asynchrony in Gibraltar Raid's GPU Reed-Solomon Coding Library. *SC Companion 2012*: pp. 1496-1497.

Xin Zhou, Anthony Skjellum, Matthew L. Curry: Poster: Evaluating Asynchrony in Gibraltar RAID's GPU Reed-Solomon Coding Library. *SC Companion 2012*: p. 1498.

* cited by examiner

FIG. 1

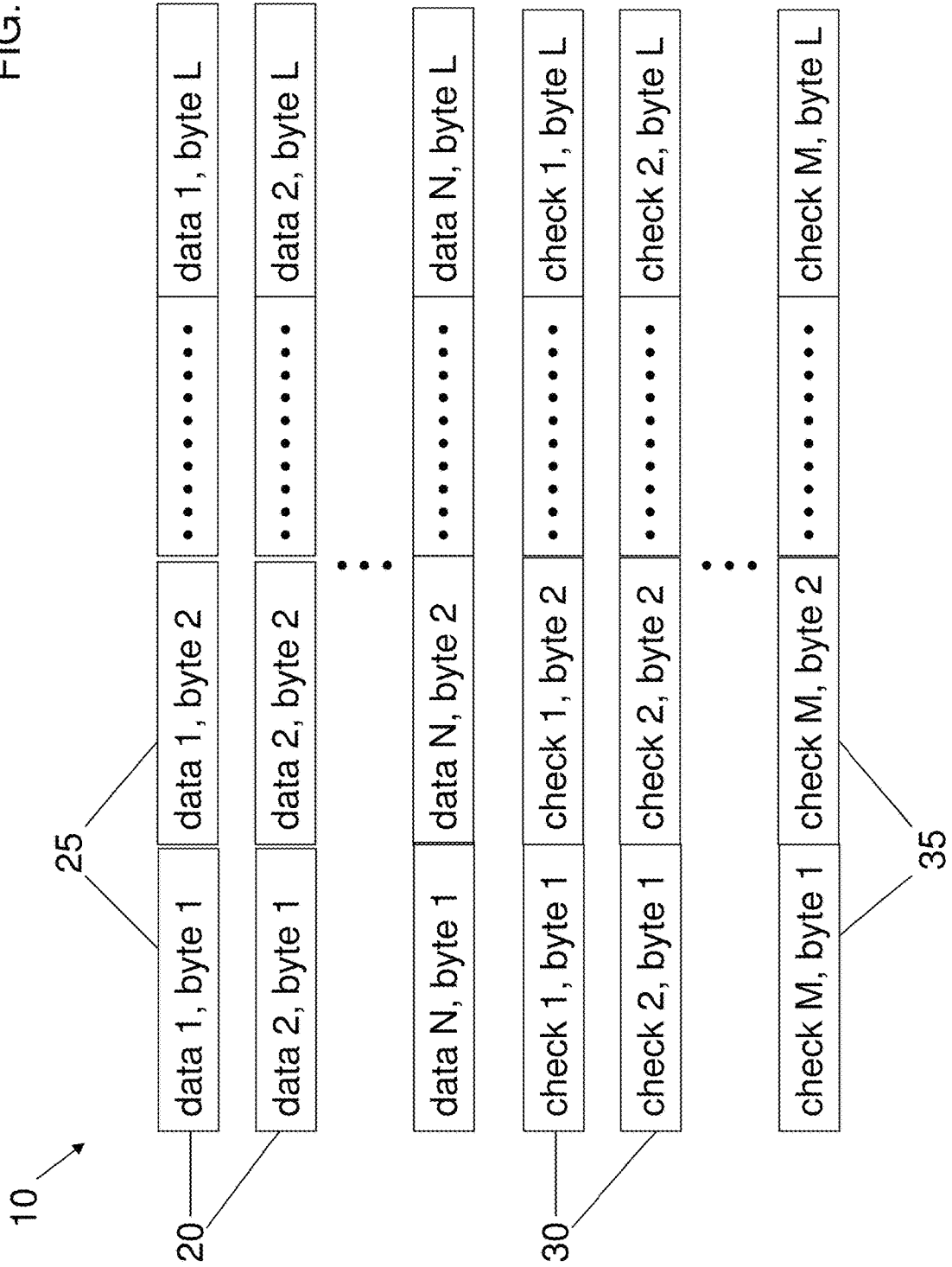


FIG. 2

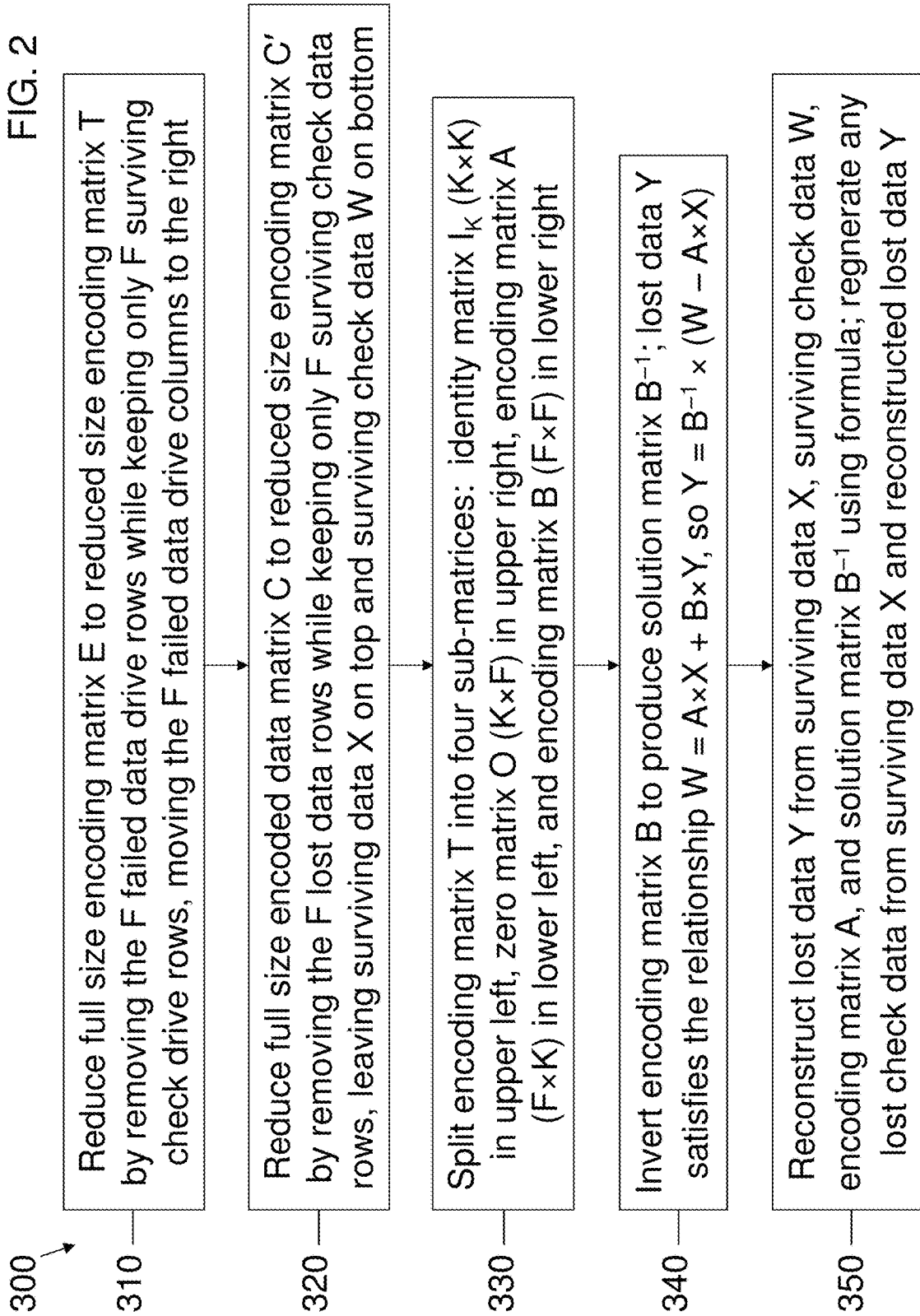


FIG. 3

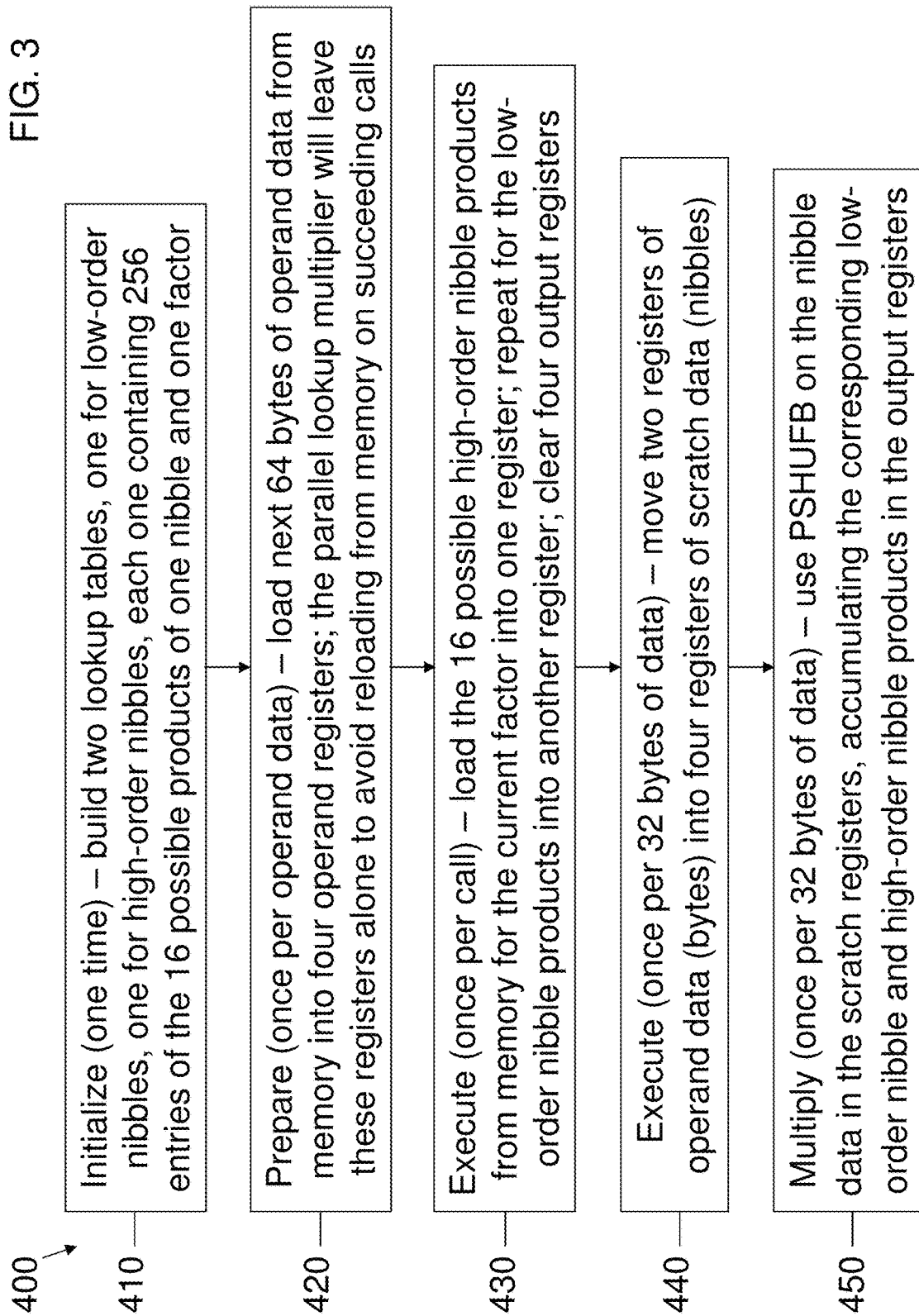


FIG. 4

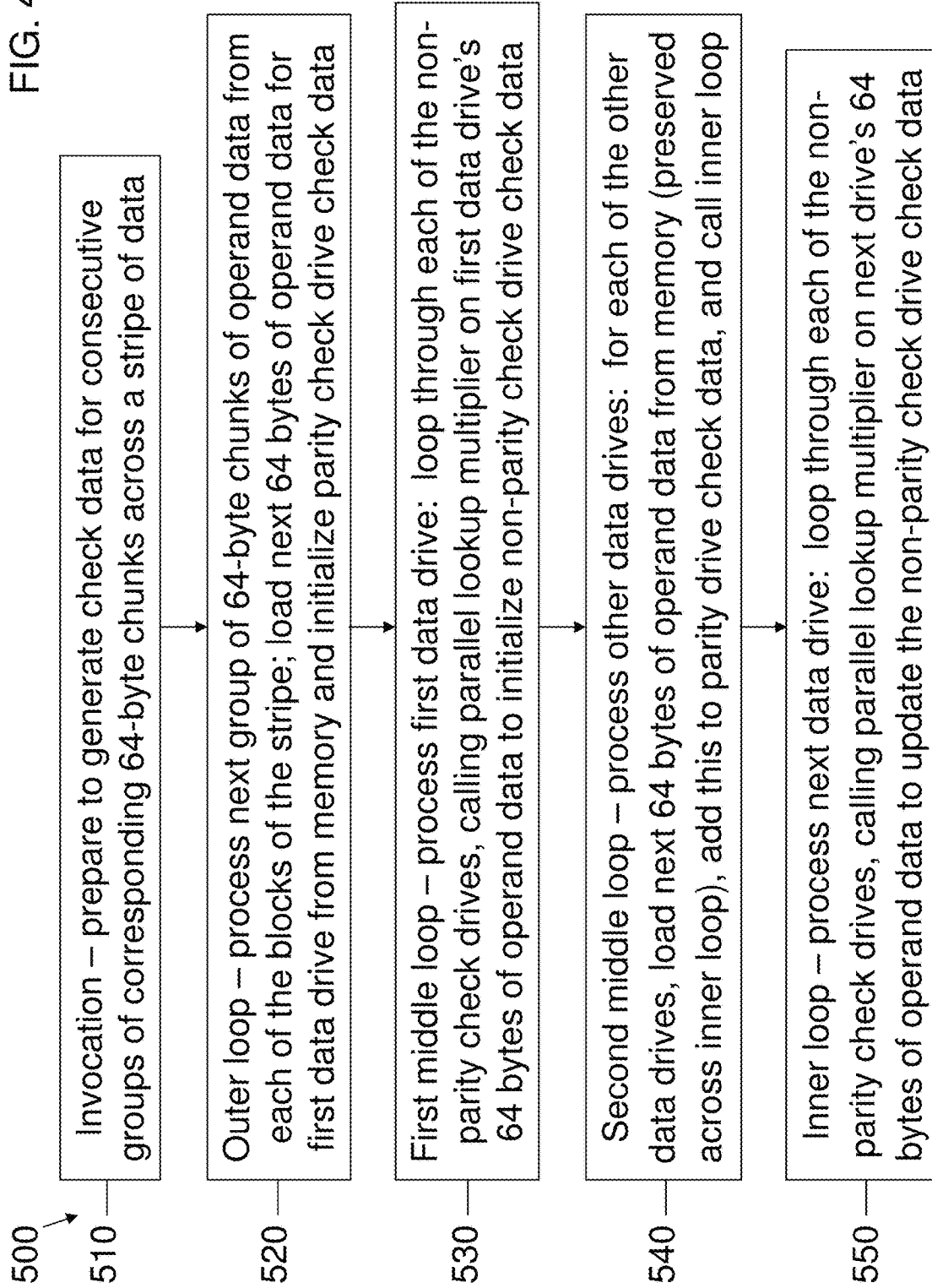


FIG. 5

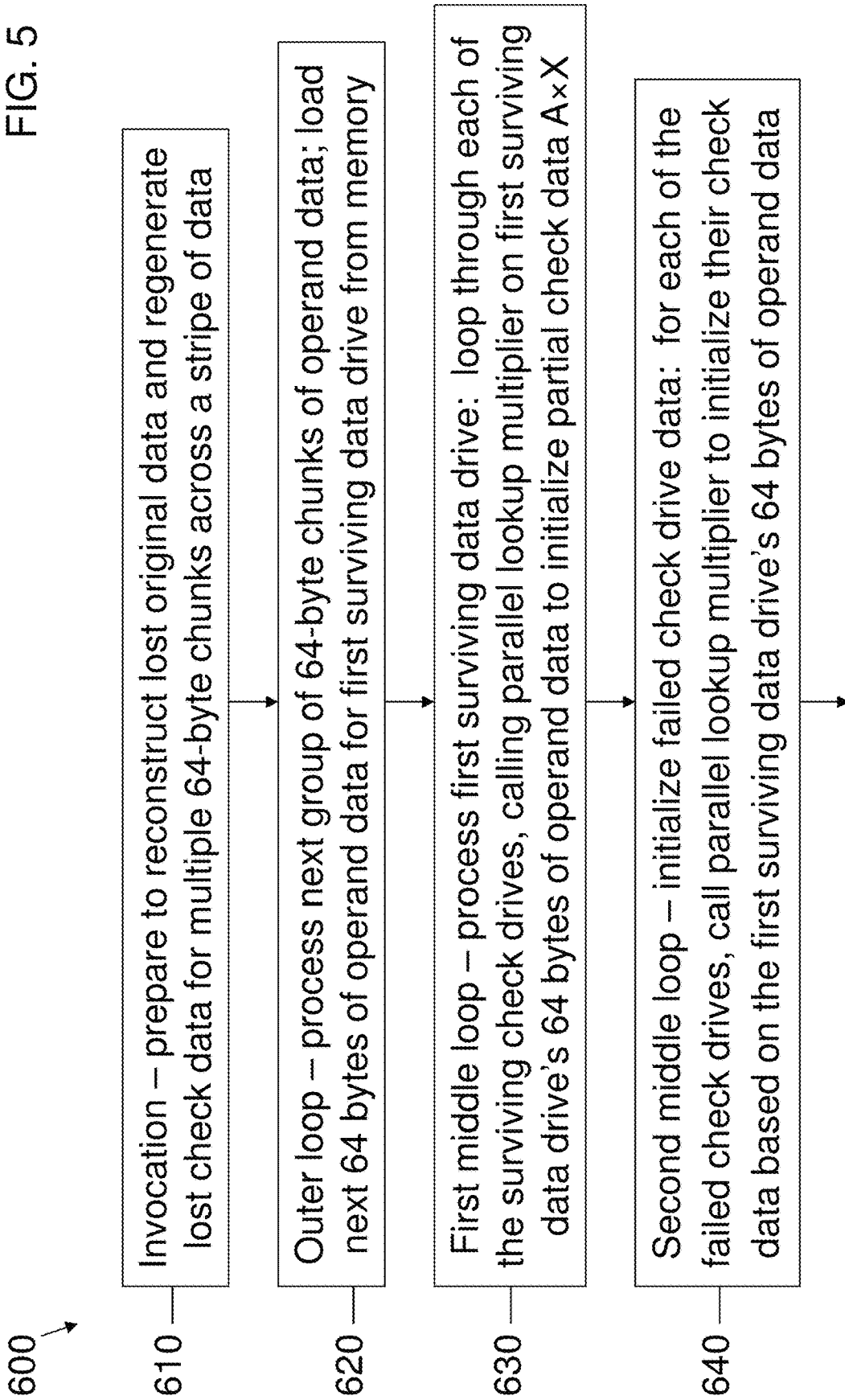


FIG. 6

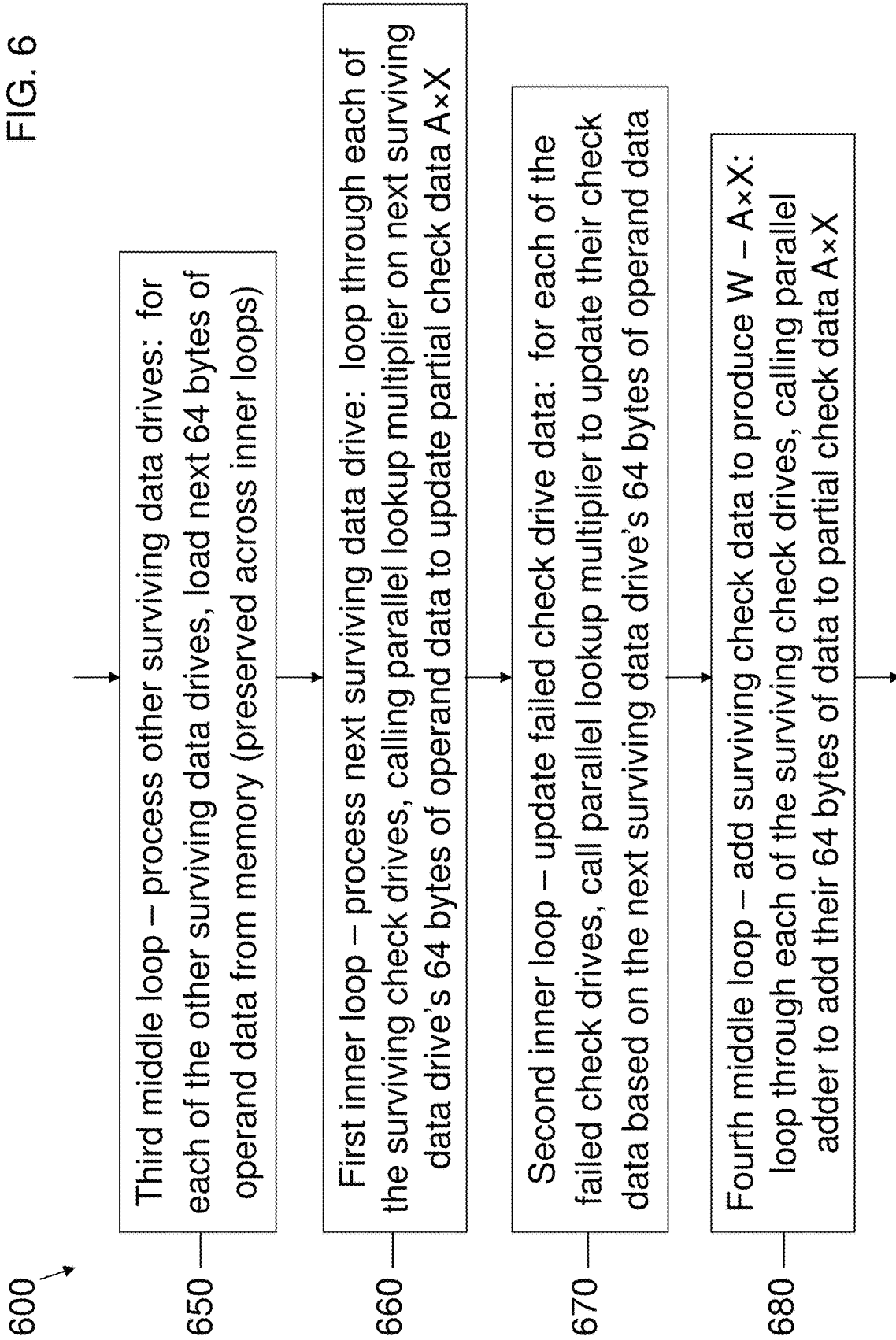


FIG. 7

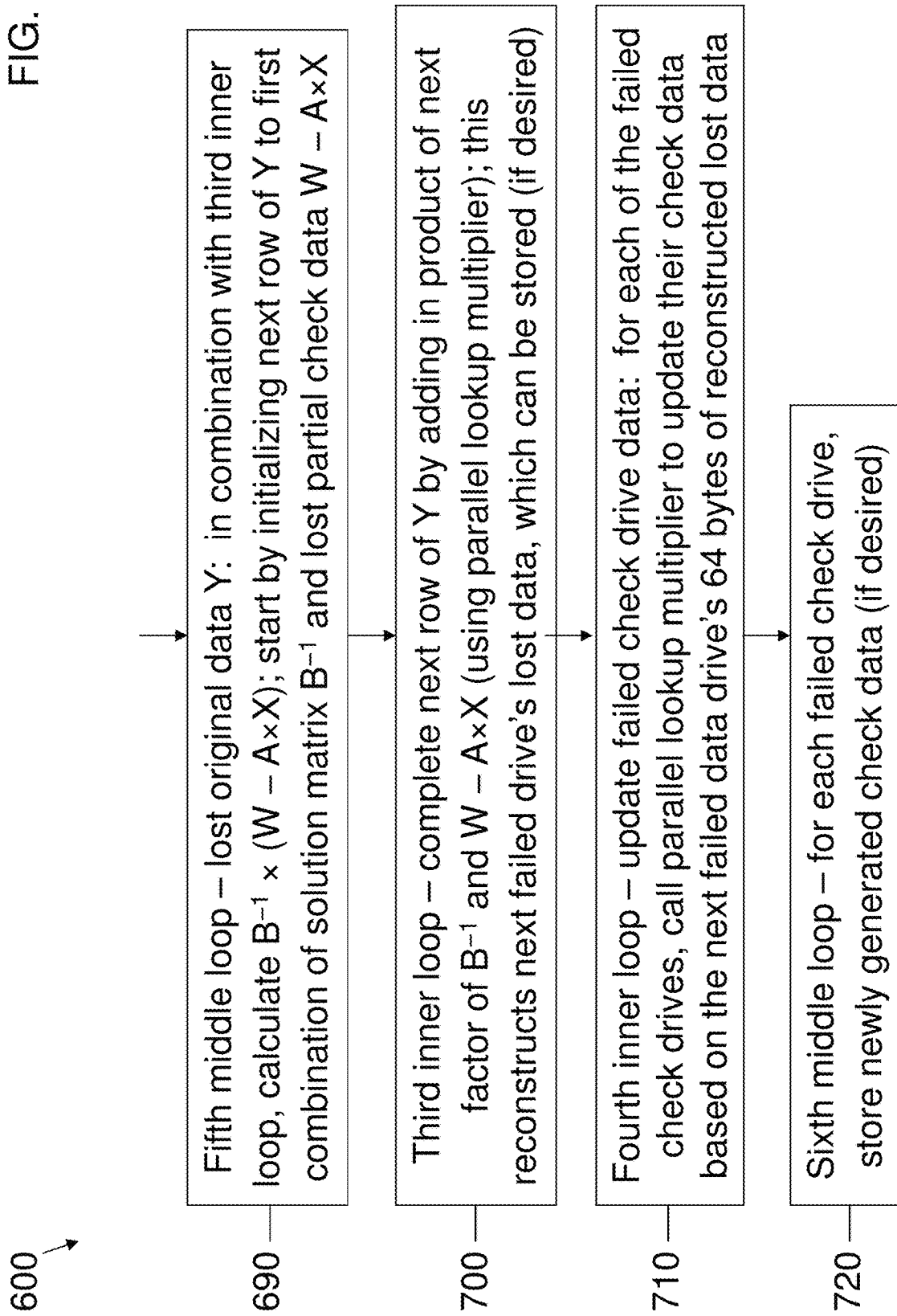


FIG. 8

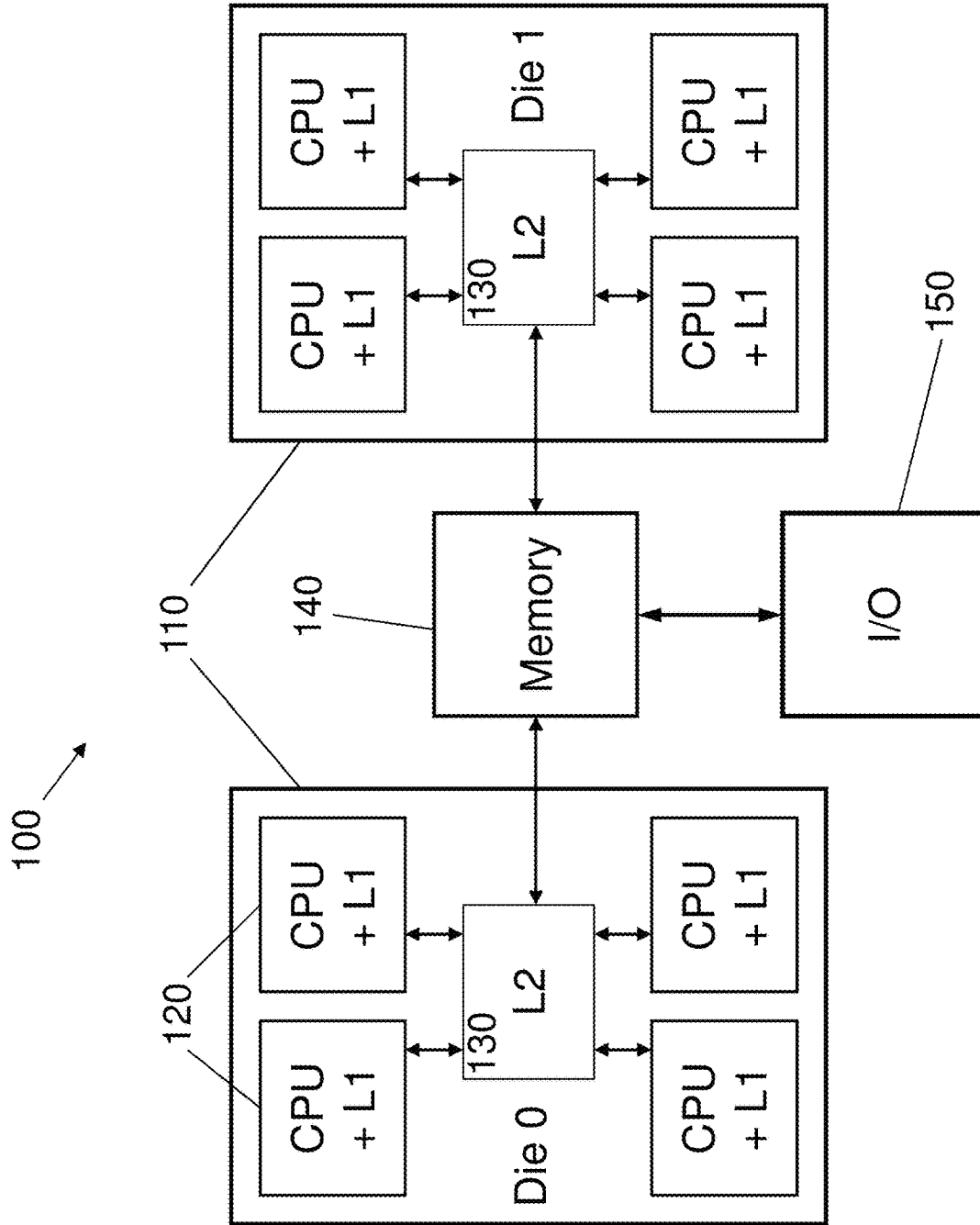
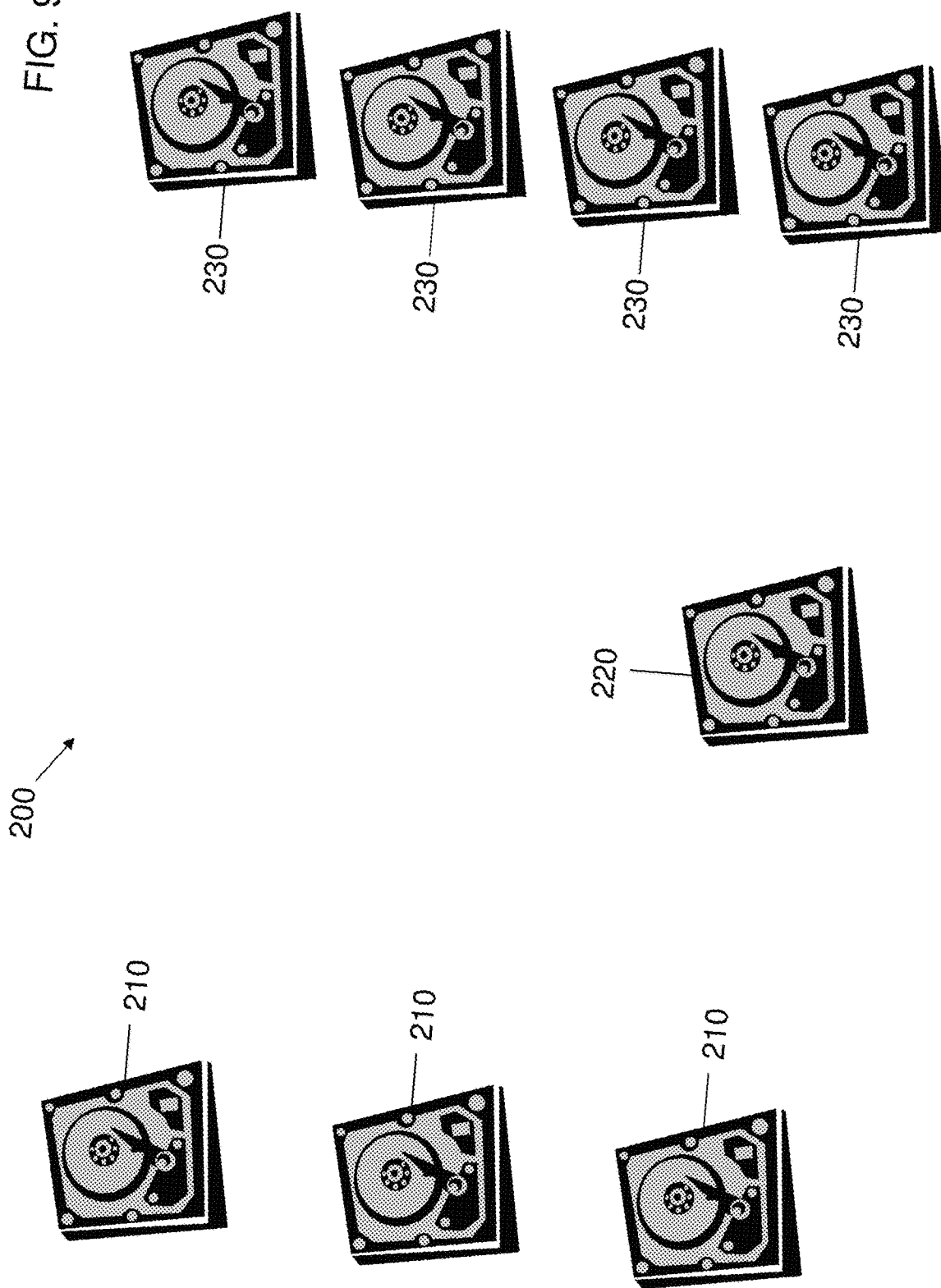


FIG. 9



US 10,291,259 B2

1

ACCELERATED ERASURE CODING SYSTEM AND METHOD

CROSS-REFERENCE TO RELATED APPLICATIONS

This application is a continuation of U.S. patent application Ser. No. 15/201,196, filed on Jul. 1, 2016, which is a continuation of U.S. patent application Ser. No. 14/852,438, filed on Sep. 11, 2015, now U.S. Pat. No. 9,385,759, issued on Jul. 5, 2016, which is a continuation of U.S. patent application Ser. No. 14/223,740, filed on Mar. 24, 2014, now U.S. Pat. No. 9,160,374, issued on Oct. 13, 2015, which is a continuation of U.S. patent application Ser. No. 13/341,833, filed on Dec. 30, 2011, now U.S. Pat. No. 8,683,296, issued on Mar. 25, 2014, the entire contents of each of which are expressly incorporated herein by reference.

BACKGROUND

Field

Aspects of embodiments of the present invention are directed toward an accelerated erasure coding system and method.

Description of Related Art

An erasure code is a type of error-correcting code (ECC) useful for forward error-correction in applications like a redundant array of independent disks (RAID) or high-speed communication systems. In a typical erasure code, data (or original data) is organized in stripes, each of which is broken up into N equal-sized blocks, or data blocks, for some positive integer N . The data for each stripe is thus reconstructable by putting the N data blocks together. However, to handle situations where one or more of the original N data blocks gets lost, erasure codes also encode an additional M equal-sized blocks (called check blocks or check data) from the original N data blocks, for some positive integer M .

The N data blocks and the M check blocks are all the same size. Accordingly, there are a total of $N+M$ equal-sized blocks after encoding. The $N+M$ blocks may, for example, be transmitted to a receiver as $N+M$ separate packets, or written to $N+M$ corresponding disk drives. For ease of description, all $N+M$ blocks after encoding will be referred to as encoded blocks, though some (for example, N of them) may contain unencoded portions of the original data. That is, the encoded data refers to the original data together with the check data.

The M check blocks build redundancy into the system, in a very efficient manner, in that the original data (as well as any lost check data) can be reconstructed if any N of the $N+M$ encoded blocks are received by the receiver, or if any N of the $N+M$ disk drives are functioning correctly. Note that such an erasure code is also referred to as “optimal.” For ease of description, only optimal erasure codes will be discussed in this application. In such a code, up to M of the encoded blocks can be lost, (e.g., up to M of the disk drives can fail) so that if any N of the $N+M$ encoded blocks are received successfully by the receiver, the original data (as well as the check data) can be reconstructed. $N/(N+M)$ is thus the code rate of the erasure code encoding (i.e., how much space the original data takes up in the encoded data). Erasure codes for select values of N and M can be implemented on RAID systems employing $N+M$ (disk) drives by spreading the original data among N “data” drives, and using the remaining M drives as “check” drives. Then, when any

2

N of the $N+M$ drives are correctly functioning, the original data can be reconstructed, and the check data can be regenerated.

Erasure codes (or more specifically, erasure coding systems) are generally regarded as impractical for values of M larger than 1 (e.g., RAID5 systems, such as parity drive systems) or 2 (RAID6 systems), that is, for more than one or two check drives. For example, see H. Peter Anvin, “The mathematics of RAID-6,” the entire content of which is incorporated herein by reference, p. 7, “Thus, in 2-disk-degraded mode, performance will be very slow. However, it is expected that that will be a rare occurrence, and that performance will not matter significantly in that case.” See also Robert Maddock et al., “Surviving Two Disk Failures,” p. 6, “The main difficulty with this technique is that calculating the check codes, and reconstructing data after failures, is quite complex. It involves polynomials and thus multiplication, and requires special hardware, or at least a signal processor, to do it at sufficient speed.” In addition, see also James S. Plank, “All About Erasure Codes: —Reed-Solomon Coding-LDPC Coding,” slide 15 (describing computational complexity of Reed-Solomon decoding), “Bottom line: When n & m grow, it is brutally expensive.” Accordingly, there appears to be a general consensus among experts in the field that erasure coding systems are impractical for RAID systems for all but small values of M (that is, small numbers of check drives), such as 1 or 2.

Modern disk drives, on the other hand, are much less reliable than those envisioned when RAID was proposed. This is due to their capacity growing out of proportion to their reliability. Accordingly, systems with only a single check disk have, for the most part, been discontinued in favor of systems with two check disks.

In terms of reliability, a higher check disk count is clearly more desirable than a lower check disk count. If the count of error events on different drives is larger than the check disk count, data may be lost and that cannot be reconstructed from the correctly functioning drives. Error events extend well beyond the traditional measure of advertised mean time between failures (MTBF). A simple, real world example is a service event on a RAID system where the operator mistakenly replaces the wrong drive or, worse yet, replaces a good drive with a broken drive. In the absence of any generally accepted methodology to train, certify, and measure the effectiveness of service technicians, these types of events occur at an unknown rate, but certainly occur. The foolproof solution for protecting data in the face of multiple error events is to increase the check disk count.

SUMMARY

Aspects of embodiments of the present invention address these problems by providing a practical erasure coding system that, for byte-level RAID processing (where each byte is made up of 8 bits), performs well even for values of $N+M$ as large as 256 drives (for example, $N=127$ data drives and $M=129$ check drives). Further aspects provide for a single precomputed encoding matrix (or master encoding matrix) S of size $M_{max} \times N_{max}$, or $(N_{max}+M_{max}) \times N_{max}$ or $(M_{max}-1) \times N_{max}$, elements (e.g., bytes), which can be used, for example, for any combination of $N \leq N_{max}$ data drives and $M \leq M_{max}$ check drives such that $N_{max}+M_{max} \leq 256$ (e.g., $N_{max}=127$ and $M_{max}=129$, or $N_{max}=63$ and $M_{max}=193$). This is an improvement over prior art solutions that rebuild such matrices from scratch every time N or M changes (such as adding another check drive). Still higher values of N and M are possible with larger processing increments, such as 2

US 10,291,259 B2

3

bytes, which affords up to $N+M=65,536$ drives (such as $N=32,767$ data drives and $M=32,769$ check drives).

Higher check disk count can offer increased reliability and decreased cost. The higher reliability comes from factors such as the ability to withstand more drive failures. The decreased cost arises from factors such as the ability to create larger groups of data drives. For example, systems with two checks disks are typically limited to group sizes of 10 or fewer drives for reliability reasons. With a higher check disk count, larger groups are available, which can lead to fewer overall components for the same unit of storage and hence, lower cost.

Additional aspects of embodiments of the present invention further address these problems by providing a standard parity drive as part of the encoding matrix. For instance, aspects provide for a parity drive for configurations with up to 127 data drives and up to 128 (non-parity) check drives, for a total of up to 256 total drives including the parity drive. Further aspects provide for different breakdowns, such as up to 63 data drives, a parity drive, and up to 192 (non-parity) check drives. Providing a parity drive offers performance comparable to RAID5 in comparable circumstances (such as single data drive failures) while also being able to tolerate significantly larger numbers of data drive failures by including additional (non-parity) check drives.

Further aspects are directed to a system and method for implementing a fast solution matrix algorithm for Reed-Solomon codes. While known solution matrix algorithms compute an $N \times N$ solution matrix (see, for example, J. S. Plank, "A tutorial on Reed-Solomon coding for fault-tolerance in RAID-like systems," *Software—Practice & Experience*, 27(9):995-1012, September 1997, and J. S. Plank and Y. Ding, "Note: Correction to the 1997 tutorial on Reed-Solomon coding," Technical Report CS-03-504, University of Tennessee, April 2003), requiring $O(N^3)$ operations, regardless of the number of failed data drives, aspects of embodiments of the present invention compute only an $F \times F$ solution matrix, where F is the number of failed data drives. The overhead for computing this $F \times F$ solution matrix is approximately $F^3/3$ multiplication operations and the same number of addition operations. Not only is $F \leq N$, in almost any practical application, the number of failed data drives F is considerably smaller than the number of data drives N . Accordingly, the fast solution matrix algorithm is considerably faster than any known approach for practical values of F and N .

Still further aspects are directed toward fast implementations of the check data generation and the lost (original and check) data reconstruction. Some of these aspects are directed toward fetching the surviving (original and check) data a minimum number of times (that is, at most once) to carry out the data reconstruction. Some of these aspects are directed toward efficient implementations that can maximize or significantly leverage the available parallel processing power of multiple cores working concurrently on the check data generation and the lost data reconstruction. Existing implementations do not attempt to accelerate these aspects of the data generation and thus fail to achieve a comparable level of performance.

In an exemplary embodiment of the present invention, a system for accelerated error-correcting code (ECC) processing is provided. The system includes a processing core for executing computer instructions and accessing data from a main memory; and a non-volatile storage medium (for example, a disk drive, or flash memory) for storing the computer instructions. The processing core, the storage medium, and the computer instructions are configured to

4

implement an erasure coding system. The erasure coding system includes a data matrix for holding original data in the main memory, a check matrix for holding check data in the main memory, an encoding matrix for holding first factors in the main memory, and a thread for executing on the processing core. The first factors are for encoding the original data into the check data. The thread includes a parallel multiplier for concurrently multiplying multiple data entries of a matrix by a single factor; and a first sequencer for ordering operations through the data matrix and the encoding matrix using the parallel multiplier to generate the check data.

The first sequencer may be configured to access each entry of the data matrix from the main memory at most once while generating the check data.

The processing core may include a plurality of processing cores. The thread may include a plurality of threads. The erasure coding system may further include a scheduler for generating the check data by dividing the data matrix into a plurality of data matrices, dividing the check matrix into a plurality of check matrices, assigning corresponding ones of the data matrices and the check matrices to the threads, and assigning the threads to the processing cores to concurrently generate portions of the check data corresponding to the check matrices from respective ones of the data matrices.

The data matrix may include a first number of rows. The check matrix may include a second number of rows. The encoding matrix may include the second number of rows and the first number of columns.

The data matrix may be configured to add rows to the first number of rows or the check matrix may be configured to add rows to the second number of rows while the first factors remain unchanged.

Each of entries of one of the rows of the encoding matrix may include a multiplicative identity factor (such as 1).

The data matrix may be configured to be divided by rows into a surviving data matrix for holding surviving original data of the original data, and a lost data matrix corresponding to lost original data of the original data and including a third number of rows. The erasure coding system may further include a solution matrix for holding second factors in the main memory. The second factors are for decoding the check data into the lost original data using the surviving original data and the first factors.

The solution matrix may include the third number of rows and the third number of columns.

The solution matrix may further include an inverted said third number by said third number sub-matrix of the encoding matrix.

The erasure coding system may further include a first list of rows of the data matrix corresponding to the surviving data matrix, and a second list of rows of the data matrix corresponding to the lost data matrix.

The data matrix may be configured to be divided into a surviving data matrix for holding surviving original data of the original data, and a lost data matrix corresponding to lost original data of the original data. The erasure coding system may further include a solution matrix for holding second factors in the main memory. The second factors are for decoding the check data into the lost original data using the surviving original data and the first factors. The thread may further include a second sequencer for ordering operations through the surviving data matrix, the encoding matrix, the check matrix, and the solution matrix using the parallel multiplier to reconstruct the lost original data.

US 10,291,259 B2

5

The second sequencer may be further configured to access each entry of the surviving data matrix from the main memory at most once while reconstructing the lost original data.

The processing core may include a plurality of processing cores. The thread may include a plurality of threads. The erasure coding system may further include: a scheduler for generating the check data and reconstructing the lost original data by dividing the data matrix into a plurality of data matrices; dividing the surviving data matrix into a plurality of surviving data matrices; dividing the lost data matrix into a plurality of lost data matrices; dividing the check matrix into a plurality of check matrices; assigning corresponding ones of the data matrices, the surviving data matrices, the lost data matrices, and the check matrices to the threads; and assigning the threads to the processing cores to concurrently generate portions of the check data corresponding to the check matrices from respective ones of the data matrices and to concurrently reconstruct portions of the lost original data corresponding to the lost data matrices from respective ones of the surviving data matrices and the check matrices.

The check matrix may be configured to be divided into a surviving check matrix for holding surviving check data of the check data, and a lost check matrix corresponding to lost check data of the check data. The second sequencer may be configured to order operations through the surviving data matrix, the reconstructed lost original data, and the encoding matrix using the parallel multiplier to regenerate the lost check data.

The second sequencer may be further configured to reconstruct the lost original data concurrently with regenerating the lost check data.

The second sequencer may be further configured to access each entry of the surviving data matrix from the main memory at most once while reconstructing the lost original data and regenerating the lost check data.

The second sequencer may be further configured to regenerate the lost check data without accessing the reconstructed lost original data from the main memory.

The processing core may include a plurality of processing cores. The thread may include a plurality of threads. The erasure coding system may further include a scheduler for generating the check data, reconstructing the lost original data, and regenerating the lost check data by: dividing the data matrix into a plurality of data matrices; dividing the surviving data matrix into a plurality of surviving data matrices; dividing the lost data matrix into a plurality of lost data matrices; dividing the check matrix into a plurality of check matrices; dividing the surviving check matrix into a plurality of surviving check matrices; dividing the lost check matrix into a plurality of lost check matrices; assigning corresponding ones of the data matrices, the surviving data matrices, the lost data matrices, the check matrices, the surviving check matrices, and the lost check matrices to the threads; and assigning the threads to the processing cores to concurrently generate portions of the check data corresponding to the check matrices from respective ones of the data matrices, to concurrently reconstruct portions of the lost original data corresponding to the lost data matrices from respective ones of the surviving data matrices and the surviving check matrices, and to concurrently regenerate portions of the lost check data corresponding to the lost check matrices from respective ones of the surviving data matrices and respective portions of the reconstructed lost original data.

The processing core may include 16 data registers. Each of the data registers may include 16 bytes. The parallel

6

multiplier may be configured to process the data in units of at least 64 bytes spread over at least four of the data registers at a time.

Consecutive instructions to process each of the units of the data may access separate ones of the data registers to permit concurrent execution of the consecutive instructions by the processing core.

The parallel multiplier may include two lookup tables for doing concurrent multiplication of 4-bit quantities across 16 byte-sized entries using the PSHUFB (Packed Shuffle Bytes) instruction.

The parallel multiplier may be further configured to receive an input operand in four of the data registers, and return with the input operand intact in the four of the data registers.

According to another exemplary embodiment of the present invention, a method of accelerated error-correcting code (ECC) processing on a computing system is provided. The computing system includes a non-volatile storage medium (such as a disk drive or flash memory), a processing core for accessing instructions and data from a main memory, and a computer program including a plurality of computer instructions for implementing an erasure coding system. The method includes: storing the computer program on the storage medium; executing the computer instructions on the processing core; arranging original data as a data matrix in the main memory; arranging first factors as an encoding matrix in the main memory, the first factors being for encoding the original data into check data, the check data being arranged as a check matrix in the main memory; and generating the check data using a parallel multiplier for concurrently multiplying multiple data entries of a matrix by a single factor. The generating of the check data includes ordering operations through the data matrix and the encoding matrix using the parallel multiplier.

The generating of the check data may include accessing each entry of the data matrix from the main memory at most once.

The processing core may include a plurality of processing cores. The executing of the computer instructions may include executing the computer instructions on the processing cores. The method may further include scheduling the generating of the check data by: dividing the data matrix into a plurality of data matrices; dividing the check matrix into a plurality of check matrices; and assigning corresponding ones of the data matrices and the check matrices to the processing cores to concurrently generate portions of the check data corresponding to the check matrices from respective ones of the data matrices.

The method may further include: dividing the data matrix into a surviving data matrix for holding surviving original data of the original data, and a lost data matrix corresponding to lost original data of the original data; arranging second factors as a solution matrix in the main memory, the second factors being for decoding the check data into the lost original data using the surviving original data and the first factors; and reconstructing the lost original data by ordering operations through the surviving data matrix, the encoding matrix, the check matrix, and the solution matrix using the parallel multiplier.

The reconstructing of the lost original data may include accessing each entry of the surviving data matrix from the main memory at most once.

The processing core may include a plurality of processing cores. The executing of the computer instructions may include executing the computer instructions on the processing cores. The method may further include scheduling the

US 10,291,259 B2

7

generating of the check data and the reconstructing of the lost original data by: dividing the data matrix into a plurality of data matrices; dividing the surviving data matrix into a plurality of surviving data matrices; dividing the lost data matrix into a plurality of lost data matrices; dividing the check matrix into a plurality of check matrices; and assigning corresponding ones of the data matrices, the surviving data matrices, the lost data matrices, and the check matrices to the processing cores to concurrently generate portions of the check data corresponding to the check matrices from respective ones of the data matrices and to concurrently reconstruct portions of the lost original data corresponding to the lost data matrices from respective ones of the surviving data matrices and the check matrices.

The method may further include: dividing the check matrix into a surviving check matrix for holding surviving check data of the check data, and a lost check matrix corresponding to lost check data of the check data; and regenerating the lost check data by ordering operations through the surviving data matrix, the reconstructed lost original data, and the encoding matrix using the parallel multiplier.

The reconstructing of the lost original data may take place concurrently with the regenerating of the lost check data.

The reconstructing of the lost original data and the regenerating of the lost check data may include accessing each entry of the surviving data matrix from the main memory at most once.

The regenerating of the lost check data may take place without accessing the reconstructed lost original data from the main memory.

The processing core may include a plurality of processing cores. The executing of the computer instructions may include executing the computer instructions on the processing cores. The method may further include scheduling the generating of the check data, the reconstructing of the lost original data, and the regenerating of the lost check data by: dividing the data matrix into a plurality of data matrices; dividing the surviving data matrix into a plurality of surviving data matrices; dividing the lost data matrix into a plurality of lost data matrices; dividing the check matrix into a plurality of check matrices; dividing the surviving check matrix into a plurality of surviving check matrices; dividing the lost check matrix into a plurality of lost check matrices; and assigning corresponding ones of the data matrices, the surviving data matrices, the lost data matrices, the check matrices, the surviving check matrices, and the lost check matrices to the processing cores to concurrently generate portions of the check data corresponding to the check matrices from respective ones of the data matrices, to concurrently reconstruct portions of the lost original data corresponding to the lost data matrices from respective ones of the surviving data matrices and the surviving check matrices, and to concurrently regenerate portions of the lost check data corresponding to the lost check matrices from respective ones of the surviving data matrices and respective portions of the reconstructed lost original data.

According to yet another exemplary embodiment of the present invention, a non-transitory computer-readable storage medium (such as a disk drive, a compact disk (CD), a digital video disk (DVD), flash memory, a universal serial bus (USB) drive, etc.) containing a computer program including a plurality of computer instructions for performing accelerated error-correcting code (ECC) processing on a computing system is provided. The computing system includes a processing core for accessing instructions and data from a main memory. The computer instructions are

8

configured to implement an erasure coding system when executed on the computing system by performing the steps of: arranging original data as a data matrix in the main memory; arranging first factors as an encoding matrix in the main memory, the first factors being for encoding the original data into check data, the check data being arranged as a check matrix in the main memory; and generating the check data using a parallel multiplier for concurrently multiplying multiple data entries of a matrix by a single factor. The generating of the check data includes ordering operations through the data matrix and the encoding matrix using the parallel multiplier.

The generating of the check data may include accessing each entry of the data matrix from the main memory at most once.

The processing core may include a plurality of processing cores. The computer instructions may be further configured to perform the step of scheduling the generating of the check data by: dividing the data matrix into a plurality of data matrices; dividing the check matrix into a plurality of check matrices; and assigning corresponding ones of the data matrices and the check matrices to the processing cores to concurrently generate portions of the check data corresponding to the check matrices from respective ones of the data matrices.

The computer instructions may be further configured to perform the steps of: dividing the data matrix into a surviving data matrix for holding surviving original data of the original data, and a lost data matrix corresponding to lost original data of the original data; arranging second factors as a solution matrix in the main memory, the second factors being for decoding the check data into the lost original data using the surviving original data and the first factors; and reconstructing the lost original data by ordering operations through the surviving data matrix, the encoding matrix, the check matrix, and the solution matrix using the parallel multiplier.

The computer instructions may be further configured to perform the steps of: dividing the check matrix into a surviving check matrix for holding surviving check data of the check data, and a lost check matrix corresponding to lost check data of the check data; and regenerating the lost check data by ordering operations through the surviving data matrix, the reconstructed lost original data, and the encoding matrix using the parallel multiplier.

The reconstructing of the lost original data and the regenerating of the lost check data may include accessing each entry of the surviving data matrix from the main memory at most once.

The processing core may include a plurality of processing cores. The computer instructions may be further configured to perform the step of scheduling the generating of the check data, the reconstructing of the lost original data, and the regenerating of the lost check data by: dividing the data matrix into a plurality of data matrices; dividing the surviving data matrix into a plurality of surviving data matrices; dividing the lost data matrix into a plurality of lost data matrices; dividing the check matrix into a plurality of check matrices; dividing the surviving check matrix into a plurality of surviving check matrices; dividing the lost check matrix into a plurality of lost check matrices; and assigning corresponding ones of the data matrices, the surviving data matrices, the lost data matrices, the check matrices, the surviving check matrices, and the lost check matrices to the processing cores to concurrently generate portions of the check data corresponding to the check matrices from respective ones of the data matrices, to concurrently reconstruct

US 10,291,259 B2

9

portions of the lost original data corresponding to the lost data matrices from respective ones of the surviving data matrices and the surviving check matrices, and to concurrently regenerate portions of the lost check data corresponding to the lost check matrices from respective ones of the surviving data matrices and respective portions of the reconstructed lost original data.

By providing practical and efficient systems and methods for erasure coding systems (which for byte-level processing can support up to $N+M=256$ drives, such as $N=127$ data drives and $M=129$ check drives, including a parity drive), applications such as RAID systems that can tolerate far more failing drives than was thought to be possible or practical can be implemented with accelerated performance significantly better than any prior art solution.

BRIEF DESCRIPTION OF THE DRAWINGS

The accompanying drawings, together with the specification, illustrate exemplary embodiments of the present invention and, together with the description, serve to explain aspects and principles of the present invention.

FIG. 1 shows an exemplary stripe of original and check data according to an embodiment of the present invention.

FIG. 2 shows an exemplary method for reconstructing lost data after a failure of one or more drives according to an embodiment of the present invention.

FIG. 3 shows an exemplary method for performing a parallel lookup Galois field multiplication according to an embodiment of the present invention.

FIG. 4 shows an exemplary method for sequencing the parallel lookup multiplier to perform the check data generation according to an embodiment of the present invention.

FIGS. 5-7 show an exemplary method for sequencing the parallel lookup multiplier to perform the lost data reconstruction according to an embodiment of the present invention.

FIG. 8 illustrates a multi-core architecture system according to an embodiment of the present invention.

FIG. 9 shows an exemplary disk drive configuration according to an embodiment of the present invention.

DETAILED DESCRIPTION

Hereinafter, exemplary embodiments of the invention will be described in more detail with reference to the accompanying drawings. In the drawings, like reference numerals refer to like elements throughout.

While optimal erasure codes have many applications, for ease of description, they will be described in this application with respect to RAID applications, i.e., erasure coding systems for the storage and retrieval of digital data distributed across numerous storage devices (or drives), though the present application is not limited thereto. For further ease of description, the storage devices will be assumed to be disk drives, though the invention is not limited thereto. In RAID systems, the data (or original data) is broken up into stripes, each of which includes N uniformly sized blocks (data blocks), and the N blocks are written across N separate drives (the data drives), one block per data drive.

In addition, for ease of description, blocks will be assumed to be composed of L elements, each element having a fixed size, say 8 bits or one byte. An element, such as a byte, forms the fundamental unit of operation for the RAID processing, but the invention is just as applicable to other size elements, such as 16 bits (2 bytes). For simplification, unless otherwise indicated, elements will be assumed to be

10

one byte in size throughout the description that follows, and the term “element(s)” and “byte(s)” will be used synonymously.

Conceptually, different stripes can distribute their data blocks across different combinations of drives, or have different block sizes or numbers of blocks, etc., but for simplification and ease of description and implementation, the described embodiments in the present application assume a consistent block size (L bytes) and distribution of blocks among the data drives between stripes. Further, all variables, such as the number of data drives N , will be assumed to be positive integers unless otherwise specified. In addition, since the $N=1$ case reduces to simple data mirroring (that is, copying the same data drive multiple times), it will also be assumed for simplicity that $N \geq 2$ throughout.

The N data blocks from each stripe are combined using arithmetic operations (to be described in more detail below) in M different ways to produce M blocks of check data (check blocks), and the M check blocks written across M drives (the check drives) separate from the N data drives, one block per check drive. These combinations can take place, for example, when new (or changed) data is written to (or back to) disk. Accordingly, each of the $N+M$ drives (data drives and check drives) stores a similar amount of data, namely one block for each stripe. As the processing of multiple stripes is conceptually similar to the processing of one stripe (only processing multiple blocks per drive instead of one), it will be further assumed for simplification that the data being stored or retrieved is only one stripe in size unless otherwise indicated. It will also be assumed that the block size L is sufficiently large that the data can be consistently divided across each block to produce subsets of the data that include respective portions of the blocks (for efficient concurrent processing by different processing units).

FIG. 1 shows an exemplary stripe **10** of original and check data according to an embodiment of the present invention.

Referring to FIG. 1, the stripe **10** can be thought of not only as the original N data blocks **20** that make up the original data, but also the corresponding M check blocks **30** generated from the original data (that is, the stripe **10** represents encoded data). Each of the N data blocks **20** is composed of L bytes **25** (labeled byte **1**, byte **2**, . . . , byte L), and each of the M check blocks **30** is composed of L bytes **35** (labeled similarly). In addition, check drive **1**, byte **1**, is a linear combination of data drive **1**, byte **1**; data drive **2**, byte **1**; . . . ; data drive N , byte **1**. Likewise, check drive **1**, byte **2**, is generated from the same linear combination formula as check drive **1**, byte **1**, only using data drive **1**, byte **2**; data drive **2**, byte **2**; . . . ; data drive N , byte **2**. In contrast, check drive **2**, byte **1**, uses a different linear combination formula than check drive **1**, byte **1**, but applies it to the same data, namely data drive **1**, byte **1**; data drive **2**, byte **1**; . . . ; data drive N , byte **1**. In this fashion, each of the other check bytes **35** is a linear combination of the respective bytes of each of the N data drives **20** and using the corresponding linear combination formula for the particular check drive **30**.

The stripe **10** in FIG. 1 can also be represented as a matrix C of encoded data. C has two sub-matrices, namely original data D on top and check data J on bottom. That is,

US 10,291,259 B2

11

$$C = \begin{bmatrix} D \\ J \end{bmatrix} = \begin{bmatrix} D_{11} & D_{12} & \dots & D_{1L} \\ D_{21} & D_{22} & \dots & D_{2L} \\ \vdots & \vdots & \ddots & \vdots \\ D_{N1} & D_{N2} & \dots & D_{NL} \\ J_{11} & J_{12} & \dots & J_{1L} \\ J_{21} & J_{22} & \dots & J_{2L} \\ \vdots & \vdots & \ddots & \vdots \\ J_{M1} & J_{M2} & \dots & J_{ML} \end{bmatrix},$$

where D_{ij} =byte j from data drive i and J_{ij} =byte j from check drive i. Thus, the rows of encoded data C represent blocks, while the columns represent corresponding bytes of each of the drives.

Further, in case of a disk drive failure of one or more disks, the arithmetic operations are designed in such a fashion that for any stripe, the original data (and by extension, the check data) can be reconstructed from any combination of N data and check blocks from the corresponding N+M data and check blocks that comprise the stripe. Thus, RAID provides both parallel processing (reading and writing the data in stripes across multiple drives concurrently) and fault tolerance (regeneration of the original data even if as many as M of the drives fail), at the computational cost of generating the check data any time new data is written to disk, or changed data is written back to disk, as well as the computational cost of reconstructing any lost original data and regenerating any lost check data after a disk failure.

For example, for M=1 check drive, a single parity drive can function as the check drive (i.e., a RAID4 system). Here, the arithmetic operation is bitwise exclusive OR of each of the N corresponding data bytes in each data block of the stripe. In addition, as mentioned earlier, the assignment of parity blocks from different stripes to the same drive (i.e., RAID4) or different drives (i.e., RAID5) is arbitrary, but it does simplify the description and implementation to use a consistent assignment between stripes, so that will be assumed throughout. Since M=1 reduces to the case of a single parity drive, it will further be assumed for simplicity that $M \geq 2$ throughout.

For such larger values of M, Galois field arithmetic is used to manipulate the data, as described in more detail later. Galois field arithmetic, for Galois fields of powers-of-2 (such as 2^p) numbers of elements, includes two fundamental operations: (1) addition (which is just bitwise exclusive OR, as with the parity drive-only operations for M=1), and (2) multiplication. While Galois field (GF) addition is trivial on standard processors, GF multiplication is not. Accordingly, a significant component of RAID performance for $M \geq 2$ is speeding up the performance of GF multiplication, as will be discussed later. For purposes of description, GF addition will be represented by the symbol + throughout while GF multiplication will be represented by the symbol x throughout.

Briefly, in exemplary embodiments of the present invention, each of the M check drives holds linear combinations (over GF arithmetic) of the N data drives of original data, one linear combination (i.e., a GF sum of N terms, where each term represents a byte of original data times a corresponding factor (using GF multiplication) for the respective data drive) for each check drive, as applied to respective bytes in each block. One such linear combination can be a simple parity, i.e., entirely GF addition (all factors equal 1), such as a GF sum of the first byte in each block of original data as described above.

12

The remaining M-1 linear combinations include more involved calculations that include the nontrivial GF multiplication operations (e.g., performing a GF multiplication of the first byte in each block by a corresponding factor for the respective data drive, and then performing a GF sum of all these products). These linear combinations can be represented by an (N+M)xN matrix (encoding matrix or information dispersal matrix (IDM)) E of the different factors, one factor for each combination of (data or check) drive and data drive, with one row for each of the N+M data and check drives and one column for each of the N data drives. The IDM E can also be represented as

$$\begin{bmatrix} I_N \\ H \end{bmatrix},$$

where I_N represents the NxN identity matrix (i.e., the original (unencoded) data) and H represents the MxN matrix of factors for the check drives (where each of the M rows corresponds to one of the M check drives and each of the N columns corresponds to one of the N data drives).

Thus,

$$E = \begin{bmatrix} I_N \\ H \end{bmatrix} = \begin{bmatrix} 1 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 1 \\ H_{11} & H_{12} & \dots & H_{1N} \\ H_{21} & H_{22} & \dots & H_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ H_{M1} & H_{M2} & \dots & H_{MN} \end{bmatrix},$$

where H_{ij} =factor for check drive i and data drive j. Thus, the rows of encoded data C represent blocks, while the columns represent corresponding bytes of each of the drives. In addition, check factors H, original data D, and check data J are related by the formula $J=H \times D$ (that is, matrix multiplication), or

$$\begin{bmatrix} J_{11} & J_{12} & \dots & J_{1L} \\ J_{21} & J_{22} & \dots & J_{2L} \\ \vdots & \vdots & \ddots & \vdots \\ J_{M1} & J_{M2} & \dots & J_{ML} \end{bmatrix} = \begin{bmatrix} H_{11} & H_{12} & \dots & H_{1N} \\ H_{21} & H_{22} & \dots & H_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ H_{M1} & H_{M2} & \dots & H_{MN} \end{bmatrix} \times \begin{bmatrix} D_{11} & D_{12} & \dots & D_{1L} \\ D_{21} & D_{22} & \dots & D_{2L} \\ \vdots & \vdots & \ddots & \vdots \\ D_{N1} & D_{N2} & \dots & D_{NL} \end{bmatrix},$$

where $J_{11}=(H_{11} \times D_{11})+(H_{12} \times D_{21})+ \dots +(H_{1N} \times D_{N1})$, $J_{12}=(H_{11} \times D_{12})+(H_{12} \times D_{22})+ \dots +(H_{1N} \times D_{N2})$, $J_{21}=(H_{21} \times D_{11})+(H_{22} \times D_{21})+ \dots +(H_{2N} \times D_{N1})$, and in general, $J_{ij}=(H_{i1} \times D_{1j})+(H_{i2} \times D_{2j})+ \dots +(H_{iN} \times D_{Nj})$ for $1 \leq i \leq M$ and $1 \leq j \leq L$.

Such an encoding matrix E is also referred to as an information dispersal matrix (IDM). It should be noted that matrices such as check drive encoding matrix H and identity matrix I_N also represent encoding matrices, in that they represent matrices of factors to produce linear combinations over GF arithmetic of the original data. In practice, the identity matrix I_N is trivial and may not need to be con-

structured as part of the IDM E. Only the encoding matrix E, however, will be referred to as the IDM. Methods of building an encoding matrix such as IDM E or check drive encoding matrix H are discussed below. In further embodiments of the present invention (as discussed further in Appendix A), such (N+M)×N (or M×N) matrices can be trivially constructed (or simply indexed) from a master encoding matrix S, which is composed of (N_{max}+M_{max})×N_{max} (or M_{max}×N_{max}) bytes or elements, where N_{max}+M_{max}=256 (or some other power of two) and N≤N_{max} and M≤M_{max}. For example, one such master encoding matrix S can include a 127×127 element identity matrix on top (for up to N_{max}=127 data drives), a row of 1's (for a parity drive), and a 128×127 element encoding matrix on bottom (for up to M_{max}=129 check drives, including the parity drive), for a total of N_{max}+M_{max}=256 drives.

The original data, in turn, can be represented by an N×L matrix D of bytes, each of the N rows representing the L bytes of a block of the corresponding one of the N data drives. If C represents the corresponding (N+M)×L matrix of encoded bytes (where each of the N+M rows corresponds to one of the N+M data and check drives), then C can be represented as

$$E \times D = \begin{bmatrix} I_N \\ H \end{bmatrix} \times D = \begin{bmatrix} I_N \times D \\ H \times D \end{bmatrix} = \begin{bmatrix} D \\ J \end{bmatrix},$$

where J=H×D is an M×L matrix of check data, with each of the M rows representing the L check bytes of the corresponding one of the M check drives. It should be noted that in the relationships such as C=E×D or J=H×D, x represents matrix multiplication over the Galois field (i.e., GF multiplication and GF addition being used to generate each of the entries in, for example, C or J).

In exemplary embodiments of the present invention, the first row of the check drive encoding matrix H (or the (N+1)th row of the IDM E) can be all 1's, representing the parity drive. For linear combinations involving this row, the GF multiplication can be bypassed and replaced with a GF sum of the corresponding bytes since the products are all trivial products involving the identity element 1. Accordingly, in parity drive implementations, the check drive encoding matrix H can also be thought of as an (M-1)×N matrix of non-trivial factors (that is, factors intended to be used in GF multiplication and not just GF addition).

Much of the RAID processing involves generating the check data when new or changed data is written to (or back to) disk. The other significant event for RAID processing is when one or more of the drives fail (data or check drives), or for whatever reason become unavailable. Assume that in such a failure scenario, F data drives fail and G check drives fail, where F and G are nonnegative integers. If F=0, then only check drives failed and all of the original data D survived. In this case, the lost check data can be regenerated from the original data D.

Accordingly, assume at least one data drive fails, that is, F≥1, and let K=N-F represent the number of data drives that survive. K is also a nonnegative integer. In addition, let X represent the surviving original data and Y represent the lost original data. That is, X is a K×L matrix composed of the K rows of the original data matrix D corresponding to the K surviving data drives, while Y is an F×L matrix composed of the F rows of the original data matrix D corresponding to the F failed data drives.

$$\begin{bmatrix} X \\ Y \end{bmatrix}$$

thus represents a permuted original data matrix D' (that is, the original data matrix D, only with the surviving original data X on top and the lost original data Y on bottom. It should be noted that once the lost original data Y is reconstructed, it can be combined with the surviving original data X to restore the original data D, from which the check data for any of the failed check drives can be regenerated.

It should also be noted that M-G check drives survive. In order to reconstruct the lost original data Y, enough (that is, at least N) total drives must survive. Given that K=N-F data drives survive, and that M-G check drives survive, it follows that (N-F)+(M-G)≥N must be true to reconstruct the lost original data Y. This is equivalent to F+G≤M (i.e., no more than F+G drives fail), or F≤M-G (that is, the number of failed data drives does not exceed the number of surviving check drives). It will therefore be assumed for simplicity that F≤M-G.

In the routines that follow, performance can be enhanced by prebuilding lists of the failed and surviving data and check drives (that is, four separate lists). This allows processing of the different sets of surviving and failed drives to be done more efficiently than existing solutions, which use, for example, bit vectors that have to be examined one bit at a time and often include large numbers of consecutive zeros (or ones) when ones (or zeros) are the bit values of interest.

FIG. 2 shows an exemplary method 300 for reconstructing lost data after a failure of one or more drives according to an embodiment of the present invention.

While the recovery process is described in more detail later, briefly it consists of two parts: (1) determining the solution matrix, and (2) reconstructing the lost data from the surviving data. Determining the solution matrix can be done in three steps with the following algorithm (Algorithm 1), with reference to FIG. 2:

1. (Step 310 in FIG. 2) Reducing the (M+N)×N IDM E to an N×N reduced encoding matrix T (also referred to as the transformed IDM) including the K surviving data drive rows and any F of the M-G surviving check drive rows (for instance, the first F surviving check drive rows, as these will include the parity drive if it survived; recall that F≤M-G was assumed). In addition, the columns of the reduced encoding matrix T are rearranged so that the K columns corresponding to the K surviving data drives are on the left side of the matrix and the F columns corresponding to the F failed drives are on the right side of the matrix. (Step 320) These F surviving check drives selected to rebuild the lost original data Y will henceforth be referred to as "the F surviving check drives," and their check data W will be referred to as "the surviving check data," even though M-G check drives survived. It should be noted that W is an F×L matrix composed of the F rows of the check data J corresponding to the F surviving check drives. Further, the surviving encoded data can be represented as a sub-matrix C' of the encoded data C. The surviving encoded data C' is an N×L matrix composed of the surviving original data X on top and the surviving check data W on bottom, that is,

$$C' = \begin{bmatrix} X \\ W \end{bmatrix}.$$

2. (Step 330) Splitting the reduced encoding matrix T into four sub-matrices (that are also encoding matrices): (i) a K×K identity matrix I_K (corresponding to the K

US 10,291,259 B2

15

surviving data drives) in the upper left, (ii) a $K \times F$ matrix O of zeros in the upper right, (iii) an $F \times K$ encoding matrix A in the lower left corresponding to the F surviving check drive rows and the K surviving data drive columns, and (iv) an $F \times F$ encoding matrix B in the lower right corresponding to the F surviving check drive rows and the F failed data drive columns. Thus, the reduced encoding matrix T can be represented as

$$\begin{bmatrix} I_K & O \\ A & B \end{bmatrix}$$

3. (Step 340) Calculating the inverse B^{-1} of the $F \times F$ encoding matrix B . As is shown in more detail in Appendix A, $C' = T \times D'$, or

$$\begin{bmatrix} X \\ W \end{bmatrix} = \begin{bmatrix} I_K & O \\ A & B \end{bmatrix} \times \begin{bmatrix} X \\ Y \end{bmatrix},$$

which is mathematically equivalent to $W = A \times X + B \times Y$. B^{-1} is the solution matrix, and is itself an $F \times F$ encoding matrix. Calculating the solution matrix B^{-1} thus allows the lost original data Y to be reconstructed from the encoding matrices A and B along with the surviving original data X and the surviving check data W .

The $F \times K$ encoding matrix A represents the original encoding matrix E , only limited to the K surviving data drives and the F surviving check drives. That is, each of the F rows of A represents a different one of the F surviving check drives, while each of the K columns of A represents a different one of the K surviving data drives. Thus, A provides the encoding factors needed to encode the original data for the surviving check drives, but only applied to the surviving data drives (that is, the surviving partial check data). Since the surviving original data X is available, A can be used to generate this surviving partial check data.

In similar fashion, the $F \times F$ encoding matrix B represents the original encoding matrix E , only limited to the F surviving check drives and the F failed data drives. That is, the F rows of B correspond to the same F rows of A , while each of the F columns of B represents a different one of the F failed data drives. Thus, B provides the encoding factors needed to encode the original data for the surviving check drives, but only applied to the failed data drives (that is, the lost partial check data). Since the lost original data Y is not available, B cannot be used to generate any of the lost partial check data. However, this lost partial check data can be determined from A and the surviving check data W . Since this lost partial check data represents the result of applying B to the lost original data Y , B^{-1} thus represents the necessary factors to reconstruct the lost original data Y from the lost partial check data.

It should be noted that steps 1 and 2 in Algorithm 1 above are logical, in that encoding matrices A and B (or the reduced encoding matrix T , for that matter) do not have to actually be constructed. Appropriate indexing of the IDM E (or the master encoding matrix S) can be used to obtain any of their entries. Step 3, however, is a matrix inversion over GF arithmetic and takes $O(F^3)$ operations, as discussed in more detail later. Nonetheless, this is a significant improvement over existing solutions, which require $O(N^3)$ operations,

16

since the number of failed data drives F is usually significantly less than the number of data drives N in any practical situation.

(Step 350 in FIG. 2) Once the encoding matrix A and the solution matrix B^{-1} are known, reconstructing the lost data from the surviving data (that is, the surviving original data X and the surviving check data W) can be accomplished in four steps using the following algorithm (Algorithm 2):

1. Use A and the surviving original data X (using matrix multiplication) to generate the surviving check data (i.e., $A \times X$), only limited to the K surviving data drives. Call this limited check data the surviving partial check data.
2. Subtract this surviving partial check data from the surviving check data W (using matrix subtraction, i.e., $W - A \times X$, which is just entry-by-entry GF subtraction, which is the same as GF addition for this Galois field). This generates the surviving check data, only this time limited to the F failed data drives. Call this limited check data the lost partial check data.
3. Use the solution matrix B^{-1} and the lost partial check data (using matrix multiplication, i.e., $B^{-1} \times (W - A \times X)$) to reconstruct the lost original data Y . Call this the recovered original data Y .
4. Use the corresponding rows of the IDM E (or master encoding matrix S) for each of the G failed check drives along with the original data D , as reconstructed from the surviving and recovered original data X and Y , to regenerate the lost check data (using matrix multiplication).

As will be shown in more detail later, steps 1-3 together require $O(F)$ operations times the amount of original data D to reconstruct the lost original data Y for the F failed data drives (i.e., roughly 1 operation per failed data drive per byte of original data D), which is proportionally equivalent to the $O(M)$ operations times the amount of original data D needed to generate the check data J for the M check drives (i.e., roughly 1 operation per check drive per byte of original data D). In addition, this same equivalence extends to step 4, which takes $O(G)$ operations times the amount of original data D needed to regenerate the lost check data for the G failed check drives (i.e., roughly 1 operation per failed check drive per byte of original data D). In summary, the number of operations needed to reconstruct the lost data is $O(F+G)$ times the amount of original data D (i.e., roughly 1 operation per failed drive (data or check) per byte of original data D). Since $F+G \leq M$, this means that the computational complexity of Algorithm 2 (reconstructing the lost data from the surviving data) is no more than that of generating the check data J from the original data D .

As mentioned above, for exemplary purposes and ease of description, data is assumed to be organized in 8-bit bytes, each byte capable of taking on $2^8 = 256$ possible values. Such data can be manipulated in byte-size elements using GF arithmetic for a Galois field of size $2^8 = 256$ elements. It should also be noted that the same mathematical principles apply to any power-of-two 2^P number of elements, not just 256, as Galois fields can be constructed for any integral power of a prime number. Since Galois fields are finite, and since GF operations never overflow, all results are the same size as the inputs, for example, 8 bits.

In a Galois field of a power-of-two number of elements, addition and subtraction are the same operation, namely a bitwise exclusive OR (XOR) of the two operands. This is a very fast operation to perform on any current processor. It can also be performed on multiple bytes concurrently. Since the addition and subtraction operations take place, for

example, on a byte-level basis, they can be done in parallel by using, for instance, x86 architecture Streaming SIMD Extensions (SSE) instructions (SIMD stands for single instruction, multiple data, and refers to performing the same instruction on different pieces of data, possibly concurrently), such as PXOR (Packed (bitwise) Exclusive OR).

SSE instructions can process, for example, 16-byte registers (XMM registers), and are able to process such registers as though they contain 16 separate one-byte operands (or 8 separate two-byte operands, or four separate four-byte operands, etc.) Accordingly, SSE instructions can do byte-level processing 16 times faster than when compared to processing a byte at a time. Further, there are 16 XMM registers, so dedicating four such registers for operand storage allows the data to be processed in 64-byte increments, using the other 12 registers for temporary storage. That is, individual operations can be performed as four consecutive SSE operations on the four respective registers (64 bytes), which can often allow such instructions to be efficiently pipelined and/or concurrently executed by the processor. In addition, the SSE instructions allows the same processing to be performed on different such 64-byte increments of data in parallel using different cores. Thus, using four separate cores can potentially speed up this processing by an additional factor of 4 over using a single core.

For example, a parallel adder (Parallel Adder) can be built using the 16-byte XMM registers and four consecutive PXOR instructions. Such parallel processing (that is, 64 bytes at a time with only a few machine-level instructions) for GF arithmetic is a significant improvement over doing the addition one byte at a time. Since the data is organized in blocks of any fixed number of bytes, such as 4096 bytes (4 kilobytes, or 4 KB) or 32,768 bytes (32 KB), a block can be composed of numerous such 64-byte chunks (e.g., 64 separate 64-byte chunks in 4 KB, or 512 chunks in 32 KB).

Multiplication in a Galois field is not as straightforward. While much of it is bitwise shifts and exclusive OR's (i.e., "additions") that are very fast operations, the numbers "wrap" in peculiar ways when they are shifted outside of their normal bounds (because the field has only a finite set of elements), which can slow down the calculations. This "wrapping" in the GF multiplication can be addressed in many ways. For example, the multiplication can be implemented serially (Serial Multiplier) as a loop iterating over the bits of one operand while performing the shifts, adds, and wraps on the other operand. Such processing, however, takes several machine instructions per bit for 8 separate bits. In other words, this technique requires dozens of machine instructions per byte being multiplied. This is inefficient compared to, for example, the performance of the Parallel Adder described above.

For another approach (Serial Lookup Multiplier), multiplication tables (of all the possible products, or at least all the non-trivial products) can be pre-computed and built ahead of time. For example, a table of $256 \times 256 = 65,536$ bytes can hold all the possible products of the two different one-byte operands). However, such tables can force serialized access on what are only byte-level operations, and not take advantage of wide (concurrent) data paths available on modern processors, such as those used to implement the Parallel Adder above.

In still another approach (Parallel Multiplier), the GF multiplication can be done on multiple bytes at a time, since the same factor in the encoding matrix is multiplied with every element in a data block. Thus, the same factor can be multiplied with 64 consecutive data block bytes at a time. This is similar to the Parallel Adder described above, only

there are several more operations needed to perform the operation. While this can be implemented as a loop on each bit of the factor, as described above, only performing the shifts, adds, and wraps on 64 bytes at a time, it can be more efficient to process the 256 possible factors as a (C language) switch statement, with inline code for each of 256 different combinations of two primitive GF operations: Multiply-by-2 and Add. For example, GF multiplication by the factor 3 can be effected by first doing a Multiply-by-2 followed by an Add. Likewise, GF multiplication by 4 is just a Multiply-by-2 followed by a Multiply-by-2 while multiplication by 6 is a Multiply-by-2 followed by an Add and then by another Multiply-by-2.

While this Add is identical to the Parallel Adder described above (e.g., four consecutive PXOR instructions to process 64 separate bytes), Multiply-by-2 is not as straightforward. For example, Multiply-by-2 in GF arithmetic can be implemented across 64 bytes at a time in 4 XMM registers via 4 consecutive PXOR instructions, 4 consecutive PCMPGTB (Packed Compare for Greater Than) instructions, 4 consecutive PADDB (Packed Add) instructions, 4 consecutive PAND (Bitwise AND) instructions, and 4 consecutive PXOR instructions. Though this takes 20 machine instructions, the instructions are very fast and results in 64 consecutive bytes of data at a time being multiplied by 2.

For 64 bytes of data, assuming a random factor between 0 and 255, the total overhead for the Parallel Multiplier is about 6 calls to multiply-by-2 and about 3.5 calls to add, or about $6 \times 20 + 3.5 \times 4 = 134$ machine instructions, or a little over 2 machine instructions per byte of data. While this compares favorably with byte-level processing, it is still possible to improve on this by building a parallel multiplier with a table lookup (Parallel Lookup Multiplier) using the PSHUFB (Packed Shuffle Bytes) instruction and doing the GF multiplication in 4-bit nibbles (half bytes).

FIG. 3 shows an exemplary method 400 for performing a parallel lookup Galois field multiplication according to an embodiment of the present invention.

Referring to FIG. 3, in step 410, two lookup tables are built once: one lookup table for the low-order nibbles in each byte, and one lookup table for the high-order nibbles in each byte. Each lookup table contains 256 sets (one for each possible factor) of the 16 possible GF products of that factor and the 16 possible nibble values. Each lookup table is thus $256 \times 16 = 4096$ bytes, which is considerably smaller than the 65,536 bytes needed to store a complete one-byte multiplication table. In addition, PSHUFB does 16 separate table lookups at once, each for one nibble, so 8 PSHUFB instructions can be used to do all the table lookups for 64 bytes (128 nibbles).

Next, in step 420, the Parallel Lookup Multiplier is initialized for the next set of 64 bytes of operand data (such as original data or surviving original data). In order to save loading this data from memory on succeeding calls, the Parallel Lookup Multiplier dedicates four registers for this data, which are left intact upon exit of the Parallel Lookup Multiplier. This allows the same data to be called with different factors (such as processing the same data for another check drive).

Next in step 430, to process these 64 bytes of operand data, the Parallel Lookup Multiplier can be implemented with 2 MOVDQA (Move Double Quadword Aligned) instructions (from memory) to do the two table lookups and 4 MOVDQA instructions (register to register) to initialize registers (such as the output registers). These are followed in steps 440 and 450 by two nearly identical sets of 17 register-to-register instructions to carry out the multiplica-

US 10,291,259 B2

19

tion 32 bytes at a time. Each such set starts (in step 440) with 5 more MOVDQA instructions for further initialization, followed by 2 PSRLW (Packed Shift Right Logical Word) instructions to realign the high-order nibbles for PSHUFB, and 4 PAND instructions to clear the high-order nibbles for PSHUFB. That is, two registers of byte operands are converted into four registers of nibble operands. Then, in step 450, 4 PSHUFB instructions are used to do the parallel table lookups, and 2 PXOR instructions to add the results of the multiplication on the two nibbles to the output registers.

Thus, the Parallel Lookup Multiplier uses 40 machine instructions to perform the parallel multiplication on 64 separate bytes, which is considerably better than the average 134 instructions for the Parallel Multiplier above, and only 10 times as many instructions as needed for the Parallel Adder. While some of the Parallel Lookup Multiplier's instructions are more complex than those of the Parallel Adder, much of this complexity can be concealed through the pipelined and/or concurrent execution of numerous such contiguous instructions (accessing different registers) on modern pipelined processors. For example, in exemplary implementations, the Parallel Lookup Multiplier has been timed at about 15 CPU clock cycles per 64 bytes processed per CPU core (about 0.36 clock cycles per instruction). In addition, the code footprint is practically nonexistent for the Parallel Lookup Multiplier (40 instructions) compared to that of the Parallel Multiplier (about 34,300 instructions), even when factoring the 8 KB needed for the two lookup tables in the Parallel Lookup Multiplier.

In addition, embodiments of the Parallel Lookup Multiplier can be passed 64 bytes of operand data (such as the next 64 bytes of surviving original data X to be processed) in four consecutive registers, whose contents can be preserved upon exiting the Parallel Lookup Multiplier (and all in the same 40 machine instructions) such that the Parallel Lookup Multiplier can be invoked again on the same 64 bytes of data without having to access main memory to reload the data. Through such a protocol, memory accesses can be minimized (or significantly reduced) for accessing the original data D during check data generation or the surviving original data X during lost data reconstruction.

Further embodiments of the present invention are directed towards sequencing this parallel multiplication (and other GF) operations. While the Parallel Lookup Multiplier processes a GF multiplication of 64 bytes of contiguous data times a specified factor, the calls to the Parallel Lookup Multiplier should be appropriately sequenced to provide efficient processing. One such sequencer (Sequencer 1), for example, can generate the check data J from the original data D, and is described further with respect to FIG. 4.

The parity drive does not need GF multiplication. The check data for the parity drive can be obtained, for example, by adding corresponding 64-byte chunks for each of the data drives to perform the parity operation. The Parallel Adder can do this using 4 instructions for every 64 bytes of data for each of the N data drives, or N/16 instructions per byte.

The M-1 non-parity check drives can invoke the Parallel Lookup Multiplier on each 64-byte chunk, using the appropriate factor for the particular combination of data drive and check drive. One consideration is how to handle the data access. Two possible ways are:

- 1) "column-by-column," i.e., 64 bytes for one data drive, followed by the next 64 bytes for that data drive, etc., and adding the products to the running total in memory (using the Parallel Adder) before moving onto the next row (data drive); and

20

- 2) "row-by-row," i.e., 64 bytes for one data drive, followed by the corresponding 64 bytes for the next data drive, etc., and keeping a running total using the Parallel Adder, then moving onto the next set of 64-byte chunks.

Column-by-column can be thought of as "constant factor, varying data," in that the (GF multiplication) factor usually remains the same between iterations while the (64-byte) data changes with each iteration. Conversely, row-by-row can be thought of as "constant data, varying factor," in that the data usually remains the same between iterations while the factor changes with each iteration.

Another consideration is how to handle the check drives. Two possible ways are:

- a) one at a time, i.e., generate all the check data for one check drive before moving onto the next check drive; and
- b) all at once, i.e., for each 64-byte chunk of original data, do all of the processing for each of the check drives before moving onto the next chunk of original data.

While each of these techniques performs the same basic operations (e.g., 40 instructions for every 64 bytes of data for each of the N data drives and M-1 non-parity check drives, or $5N(M-1)/8$ instructions per byte for the Parallel Lookup Multiplier), empirical results show that combination (2)(b), that is, row-by-row data access on all of the check drives between data accesses performs best with the Parallel Lookup Multiplier. One reason may be that such an approach appears to minimize the number of memory accesses (namely, one) to each chunk of the original data D to generate the check data J. This embodiment of Sequencer 1 is described in more detail with reference to FIG. 4.

FIG. 4 shows an exemplary method 500 for sequencing the Parallel Lookup Multiplier to perform the check data generation according to an embodiment of the present invention.

Referring to FIG. 4, in step 510, the Sequencer 1 is called. Sequencer 1 is called to process multiple 64-byte chunks of data for each of the blocks across a stripe of data. For instance, Sequencer 1 could be called to process 512 bytes from each block. If, for example, the block size L is 4096 bytes, then it would take eight such calls to Sequencer 1 to process the entire stripe. The other such seven calls to Sequencer 1 could be to different processing cores, for instance, to carry out the check data generation in parallel. The number of 64-byte chunks to process at a time could depend on factors such as cache dimensions, input/output data structure sizes, etc.

In step 520, the outer loop processes the next 64-byte chunk of data for each of the drives. In order to minimize the number of accesses of each data drive's 64-byte chunk of data from memory, the data is loaded only once and preserved across calls to the Parallel Lookup Multiplier. The first data drive is handled specially since the check data has to be initialized for each check drive. Using the first data drive to initialize the check data saves doing the initialization as a separate step followed by updating it with the first data drive's data. In addition to the first data drive, the first check drive is also handled specially since it is a parity drive, so its check data can be initialized to the first data drive's data directly without needing the Parallel Lookup Multiplier.

In step 530, the first middle loop is called, in which the remainder of the check drives (that is, the non-parity check drives) have their check data initialized by the first data drive's data. In this case, there is a corresponding factor (that varies with each check drive) that needs to be multiplied

US 10,291,259 B2

21

with each of the first data drive's data bytes. This is handled by calling the Parallel Lookup Multiplier for each non-parity check drive.

In step 540, the second middle loop is called, which processes the other data drives' corresponding 64-byte chunks of data. As with the first data drive, each of the other data drives is processed separately, loading the respective 64 bytes of data into four registers (preserved across calls to the Parallel Lookup Multiplier). In addition, since the first check drive is the parity drive, its check data can be updated by directly adding these 64 bytes to it (using the Parallel Adder) before handling the non-parity check drives.

In step 550, the inner loop is called for the next data drive. In the inner loop (as with the first middle loop), each of the non-parity check drives is associated with a corresponding factor for the particular data drive. The factor is multiplied with each of the next data drive's data bytes using the Parallel Lookup Multiplier, and the results added to the check drive's check data.

Another such sequencer (Sequencer 2) can be used to reconstruct the lost data from the surviving data (using Algorithm 2). While the same column-by-column and row-by-row data access approaches are possible, as well as the same choices for handling the check drives, Algorithm 2 adds another dimension of complexity because of the four separate steps and whether to: (i) do the steps completely serially or (ii) do some of the steps concurrently on the same data. For example, step 1 (surviving check data generation) and step 4 (lost check data regeneration) can be done concurrently on the same data to reduce or minimize the number of surviving original data accesses from memory.

Empirical results show that method (2)(b)(ii), that is, row-by-row data access on all of the check drives and for both surviving check data generation and lost check data regeneration between data accesses performs best with the Parallel Lookup Multiplier when reconstructing lost data using Algorithm 2. Again, this may be due to the apparent minimization of the number of memory accesses (namely, one) of each chunk of surviving original data X to reconstruct the lost data and the absence of memory accesses of reconstructed lost original data Y when regenerating the lost check data. This embodiment of Sequencer 1 is described in more detail with reference to FIGS. 5-7.

FIGS. 5-7 show an exemplary method 600 for sequencing the Parallel Lookup Multiplier to perform the lost data reconstruction according to an embodiment of the present invention.

Referring to FIG. 5, in step 610, the Sequencer 2 is called. Sequencer 2 has many similarities with the embodiment of Sequencer 1 illustrated in FIG. 4. For instance, Sequencer 2 processes the data drive data in 64-byte chunks like Sequencer 1. Sequencer 2 is more complex, however, in that only some of the data drive data is surviving; the rest has to be reconstructed. In addition, lost check data needs to be regenerated. Like Sequencer 1, Sequencer 2 does these operations in such a way as to minimize memory accesses of the data drive data (by loading the data once and calling the Parallel Lookup Multiplier multiple times). Assume for ease of description that there is at least one surviving data drive; the case of no surviving data drives is handled a little differently, but not significantly different. In addition, recall from above that the driving formula behind data reconstruction is $Y=B^{-1} \times (W-A \times X)$, where Y is the lost original data, B^{-1} is the solution matrix, W is the surviving check data, A is the partial check data encoding matrix (for the surviving check drives and the surviving data drives), and X is the surviving original data.

22

In step 620, the outer loop processes the next 64-byte chunk of data for each of the drives Like Sequencer 1, the first surviving data drive is again handled specially since the partial check data $A \times X$ has to be initialized for each surviving check drive.

In step 630, the first middle loop is called, in which the partial check data $A \times X$ is initialized for each surviving check drive based on the first surviving data drive's 64 bytes of data. In this case, the Parallel Lookup Multiplier is called for each surviving check drive with the corresponding factor (from A) for the first surviving data drive.

In step 640, the second middle loop is called, in which the lost check data is initialized for each failed check drive. Using the same 64 bytes of the first surviving data drive (preserved across the calls to Parallel Lookup Multiplier in step 630), the Parallel Lookup Multiplier is again called, this time to initialize each of the failed check drive's check data to the corresponding component from the first surviving data drive. This completes the computations involving the first surviving data drive's 64 bytes of data, which were fetched with one access from main memory and preserved in the same four registers across steps 630 and 640.

Continuing with FIG. 6, in step 650, the third middle loop is called, which processes the other surviving data drives' corresponding 64-byte chunks of data. As with the first surviving data drive, each of the other surviving data drives is processed separately, loading the respective 64 bytes of data into four registers (preserved across calls to the Parallel Lookup Multiplier).

In step 660, the first inner loop is called, in which the partial check data $A \times X$ is updated for each surviving check drive based on the next surviving data drive's 64 bytes of data. In this case, the Parallel Lookup Multiplier is called for each surviving check drive with the corresponding factor (from A) for the next surviving data drive.

In step 670, the second inner loop is called, in which the lost check data is updated for each failed check drive. Using the same 64 bytes of the next surviving data drive (preserved across the calls to Parallel Lookup Multiplier in step 660), the Parallel Lookup Multiplier is again called, this time to update each of the failed check drive's check data by the corresponding component from the next surviving data drive. This completes the computations involving the next surviving data drive's 64 bytes of data, which were fetched with one access from main memory and preserved in the same four registers across steps 660 and 670.

Next, in step 680, the computation of the partial check data $A \times X$ is complete, so the surviving check data W is added to this result (recall that $W-A \times X$ is equivalent to $W+A \times X$ in binary Galois Field arithmetic). This is done by the fourth middle loop, which for each surviving check drive adds the corresponding 64-byte component of surviving check data W to the (surviving) partial check data $A \times X$ (using the Parallel Adder) to produce the (lost) partial check data $W-A \times X$.

Continuing with FIG. 7, in step 690, the fifth middle loop is called, which performs the two dimensional matrix multiplication $B^{-1} \times (W-A \times X)$ to produce the lost original data Y. The calculation is performed one row at a time, for a total of F rows, initializing the row to the first term of the corresponding linear combination of the solution matrix B^{-1} and the lost partial check data $W-A \times X$ (using the Parallel Lookup Multiplier).

In step 700, the third inner loop is called, which completes the remaining F-1 terms of the corresponding linear combination (using the Parallel Lookup Multiplier on each term) from the fifth middle loop in step 690 and updates the

running calculation (using the Parallel Adder) of the next row of $B^{-1} \times (W - A \times X)$. This completes the next row (and reconstructs the corresponding failed data drive's lost data) of lost original data Y, which can then be stored at an appropriate location.

In step 710, the fourth inner loop is called, in which the lost check data is updated for each failed check drive by the newly reconstructed lost data for the next failed data drive. Using the same 64 bytes of the next reconstructed lost data (preserved across calls to the Parallel Lookup Multiplier), the Parallel Lookup Multiplier is called to update each of the failed check drives' check data by the corresponding component from the next failed data drive. This completes the computations involving the next failed data drive's 64 bytes of reconstructed data, which were performed as soon as the data was reconstructed and without being stored and retrieved from main memory.

Finally, in step 720, the sixth middle loop is called. The lost check data has been regenerated, so in this step, the newly regenerated check data is stored at an appropriate location (if desired).

Aspects of the present invention can be also realized in other environments, such as two-byte quantities, each such two-byte quantity capable of taking on $2^{16} = 65,536$ possible values, by using similar constructs (scaled accordingly) to those presented here. Such extensions would be readily apparent to one of ordinary skill in the art, so their details will be omitted for brevity of description.

Exemplary techniques and methods for doing the Galois field manipulation and other mathematics behind RAID error correcting codes are described in Appendix A, which contains a paper "Information Dispersal Matrices for RAID Error Correcting Codes" prepared for the present application.

Multi-core Considerations

What follows is an exemplary embodiment for optimizing or improving the performance of multi-core architecture systems when implementing the described erasure coding system routines. In multi-core architecture systems, each processor die is divided into multiple CPU cores, each with their own local caches, together with a memory (bus) interface and possible on-die cache to interface with a shared memory with other processor dies.

FIG. 8 illustrates a multi-core architecture system 100 having two processor dies 110 (namely, Die 0 and Die 1).

Referring to FIG. 8, each die 110 includes four central processing units (CPUs or cores) 120 each having a local level 1 (L1) cache. Each core 120 may have separate functional units, for example, an x86 execution unit (for traditional instructions) and a SSE execution unit (for software designed for the newer SSE instruction set). An example application of these function units is that the x86 execution unit can be used for the RAID control logic software while the SSE execution unit can be used for the GF operation software. Each die 110 also has a level 2 (L2) cache/memory bus interface 130 shared between the four cores 120. Main memory 140, in turn, is shared between the two dies 110, and is connected to the input/output (I/O) controllers 150 that access external devices such as disk drives or other non-volatile storage devices via interfaces such as Peripheral Component Interconnect (PCI).

Redundant array of independent disks (RAID) controller processing can be described as a series of states or functions. These states may include: (1) Command Processing, to validate and schedule a host request (for example, to load or store data from disk storage); (2) Command Translation and Submission, to translate the host request into multiple disk

requests and to pass the requests to the physical disks; (3) Error Correction, to generate check data and reconstruct lost data when some disks are not functioning correctly; and (4) Request Completion, to move data from internal buffers to requestor buffers. Note that the final state, Request Completion, may only be needed for a RAID controller that supports caching, and can be avoided in a cacheless design.

Parallelism is achieved in the embodiment of FIG. 8 by assigning different cores 120 to different tasks. For example, some of the cores 120 can be "command cores," that is, assigned to the I/O operations, which includes reading and storing the data and check bytes to and from memory 140 and the disk drives via the I/O interface 150. Others of the cores 120 can be "data cores," and assigned to the GF operations, that is, generating the check data from the original data, reconstructing the lost data from the surviving data, etc., including the Parallel Lookup Multiplier and the sequencers described above. For example, in exemplary embodiments, a scheduler can be used to divide the original data D into corresponding portions of each block, which can then be processed independently by different cores 120 for applications such as check data generation and lost data reconstruction.

One of the benefits of this data core/command core subdivision of processing is ensuring that different code will be executed in different cores 120 (that is, command code in command cores, and data code in data cores). This improves the performance of the associated L1 cache in each core 120, and avoids the "pollution" of these caches with code that is less frequently executed. In addition, empirical results show that the dies 110 perform best when only one core 120 on each die 110 does the GF operations (i.e., Sequencer 1 or Sequencer 2, with corresponding calls to Parallel Lookup Multiplier) and the other cores 120 do the I/O operations. This helps localize the Parallel Lookup Multiplier code and associated data to a single core 120 and not compete with other cores 120, while allowing the other cores 120 to keep the data moving between memory 140 and the disk drives via the I/O interface 150.

Embodiments of the present invention yield scalable, high performance RAID systems capable of outperforming other systems, and at much lower cost, due to the use of high volume commodity components that are leveraged to achieve the result. This combination can be achieved by utilizing the mathematical techniques and code optimizations described elsewhere in this application with careful placement of the resulting code on specific processing cores. Embodiments can also be implemented on fewer resources, such as single-core dies and/or single-die systems, with decreased parallelism and performance optimization.

The process of subdividing and assigning individual cores 120 and/or dies 110 to inherently parallelizable tasks will result in a performance benefit. For example, on a Linux system, software may be organized into "threads," and threads may be assigned to specific CPUs and memory systems via the `kthread_bind` function when the thread is created. Creating separate threads to process the GF arithmetic allows parallel computations to take place, which multiplies the performance of the system.

Further, creating multiple threads for command processing allows for fully overlapped execution of the command processing states. One way to accomplish this is to number each command, then use the arithmetic MOD function (`%` in C language) to choose a separate thread for each command. Another technique is to subdivide the data processing portion of each command into multiple components, and assign each component to a separate thread.

US 10,291,259 B2

25

FIG. 9 shows an exemplary disk drive configuration 200 according to an embodiment of the present invention.

Referring to FIG. 9, eight disks are shown, though this number can vary in other embodiments. The disks are divided into three types: data drives 210, parity drive 220, and check drives 230. The eight disks break down as three data drives 210, one parity drive 220, and four check drives 230 in the embodiment of FIG. 9.

Each of the data drives 210 is used to hold a portion of data. The data is distributed uniformly across the data drives 210 in stripes, such as 192 KB stripes. For example, the data for an application can be broken up into stripes of 192 KB, and each of the stripes in turn broken up into three 64 KB blocks, each of the three blocks being written to a different one of the three data drives 210.

The parity drive 220 is a special type of check drive in that the encoding of its data is a simple summation (recall that this is exclusive OR in binary GF arithmetic) of the corresponding bytes of each of the three data drives 210. That is, check data generation (Sequencer 1) or regeneration (Sequencer 2) can be performed for the parity drive 220 using the Parallel Adder (and not the Parallel Lookup Multiplier). Accordingly, the check data for the parity drive 220 is relatively straightforward to build. Likewise, when one of the data drives 210 no longer functions correctly, the parity drive 220 can be used to reconstruct the lost data by adding (same as subtracting in binary GF arithmetic) the corresponding bytes from each of the two remaining data drives 210. Thus, a single drive failure of one of the data drives 210 is very straightforward to handle when the parity drive 220 is available (no Parallel Lookup Multiplier). Accordingly, the parity drive 220 can replace much of the GF multiplication operations with GF addition for both check data generation and lost data reconstruction.

Each of the check drives 230 contains a linear combination of the corresponding bytes of each of the data drives 210. The linear combination is different for each check drive 230, but in general is represented by a summation of different multiples of each of the corresponding bytes of the data drives 210 (again, all arithmetic being GF arithmetic). For example, for the first check drive 230, each of the bytes of the first data drive 210 could be multiplied by 4, each of the bytes of the second data drive 210 by 3, and each of the bytes of the third data drive 210 by 6, then the corresponding products for each of the corresponding bytes could be added to produce the first check drive data. Similar linear combinations could be used to produce the check drive data for the other check drives 230. The specifics of which multiples for which check drive are explained in Appendix A.

With the addition of the parity drive 220 and check drives 230, eight drives are used in the RAID system 200 of FIG. 9. Accordingly, each 192 KB of original data is stored as 512 KB (i.e., eight blocks of 64 KB) of (original plus check) data. Such a system 200, however, is capable of recovering all of the original data provided any three of these eight drives survive. That is, the system 200 can withstand a concurrent failure of up to any five drives and still preserve all of the original data.

Exemplary Routines to Implement an Embodiment

The error correcting code (ECC) portion of an exemplary embodiment of the present invention may be written in software as, for example, four functions, which could be named as ECCInitialize, ECCSolve, ECCGenerate, and ECCRegenerate. The main functions that perform work are ECCGenerate and ECCRegenerate. ECCGenerate generates check codes for data that are used to recover data when a drive suffers an outage (that is, ECCGenerate generates the

26

check data J from the original data D using Sequencer 1). ECCRegenerate uses these check codes and the remaining data to recover data after such an outage (that is, ECCRegenerate uses the surviving check data W, the surviving original data X, and Sequencer 2 to reconstruct the lost original data Y while also regenerating any of the lost check data). Prior to calling either of these functions, ECCSolve is called to compute the constants used for a particular configuration of data drives, check drives, and failed drives (for example, ECCSolve builds the solution matrix B^{-1} together with the lists of surviving and failed data and check drives). Prior to calling ECCSolve, ECCInitialize is called to generate constant tables used by all of the other functions (for example, ECCInitialize builds the IDM E and the two lookup tables for the Parallel Lookup Multiplier).

ECCInitialize

The function ECCInitialize creates constant tables that are used by all subsequent functions. It is called once at program initialization time. By copying or precomputing these values up front, these constant tables can be used to replace more time-consuming operations with simple table look-ups (such as for the Parallel Lookup Multiplier). For example, four tables useful for speeding up the GF arithmetic include:

1. mvct—an array of constants used to perform GF multiplication with the PSHUFB instruction that operates on SSE registers (that is, the Parallel Lookup Multiplier).
2. mast—contains the master encoding matrix S (or the Information Dispersal Matrix (IDM) E, as described in Appendix A), or at least the nontrivial portion, such as the check drive encoding matrix H
3. mul_tab—contains the results of all possible GF multiplication operations of any two operands (for example, $256 \times 256 = 65,536$ bytes for all of the possible products of two different one-byte quantities)
4. div_tab—contains the results of all possible GF division operations of any two operands (can be similar in size to mul_tab)

ECCSolve

The function ECCSolve creates constant tables that are used to compute a solution for a particular configuration of data drives, check drives, and failed drives. It is called prior to using the functions ECCGenerate or ECCRegenerate. It allows the user to identify a particular case of failure by describing the logical configuration of data drives, check drives, and failed drives. It returns the constants, tables, and lists used to either generate check codes or regenerate data. For example, it can return the matrix B that needs to be inverted as well as the inverted matrix B^{-1} (i.e., the solution matrix).

ECCGenerate

The function ECCGenerate is used to generate check codes (that is, the check data matrix J) for a particular configuration of data drives and check drives, using Sequencer 1 and the Parallel Lookup Multiplier as described above. Prior to calling ECCGenerate, ECCSolve is called to compute the appropriate constants for the particular configuration of data drives and check drives, as well as the solution matrix B^{-1} .

ECCRegenerate

The function ECCRegenerate is used to regenerate data vectors and check code vectors for a particular configuration of data drives and check drives (that is, reconstructing the original data matrix D from the surviving data matrix X and the surviving check matrix W, as well as regenerating the lost check data from the restored original data), this time using Sequencer 2 and the Parallel Lookup Multiplier as

US 10,291,259 B2

27

described above. Prior to calling ECCRegenerate, ECCSolve is called to compute the appropriate constants for the particular configuration of data drives, check drives, and failed drives, as well as the solution matrix B^{-1} .

Exemplary Implementation Details

As discussed in Appendix A, there are two significant sources of computational overhead in erasure code processing (such as an erasure coding system used in RAID processing): the computation of the solution matrix B^{-1} for a given failure scenario, and the byte-level processing of encoding the check data J and reconstructing the lost data after a lost packet (e.g., data drive failure). By reducing the solution matrix B^{-1} to a matrix inversion of a $F \times F$ matrix, where F is the number of lost packets (e.g., failed drives), that portion of the computational overhead is for all intents and purposes negligible compared to the megabytes (MB), gigabytes (GB), and possibly terabytes (TB) of data that needs to be encoded into check data or reconstructed from the surviving original and check data. Accordingly, the remainder of this section will be devoted to the byte-level encoding and regenerating processing.

As already mentioned, certain practical simplifications can be assumed for most implementations. By using a Galois field of 256 entries, byte-level processing can be used for all of the GF arithmetic. Using the master encoding matrix S described in Appendix A, any combination of up to 127 data drives, 1 parity drive, and 128 check drives can be supported with such a Galois field. While, in general, any combination of data drives and check drives that adds up to 256 total drives is possible, not all combinations provide a parity drive when computed directly. Using the master encoding matrix S , on the other hand, allows all such combinations (including a parity drive) to be built (or simply indexed) from the same such matrix. That is, the appropriate sub-matrix (including the parity drive) can be used for configurations of less than the maximum number of drives.

In addition, using the master encoding matrix S permits further data drives and/or check drives can be added without requiring the recomputing of the IDM E (unlike other proposed solutions, which recompute E for every change of N or M). Rather, additional indexing of rows and/or columns of the master encoding matrix S will suffice. As discussed above, the use of the parity drive can eliminate or significantly reduce the somewhat complex GF multiplication operations associated with the other check drives and replaces them with simple GF addition (bitwise exclusive OR in binary Galois fields) operations. It should be noted that master encoding matrices with the above properties are possible for any power-of-two number of drives $2^P = N_{max} + M_{max}$, where the maximum number of data drives N_{max} is one less than a power of two (e.g., $N_{max} = 127$ or 63) and the maximum number of check drives M_{max} (including the parity drive) is $2^P - N_{max}$.

As discussed earlier, in an exemplary embodiment of the present invention, a modern x86 architecture is used (being readily available and inexpensive). In particular, this architecture supports 16 XMM registers and the SSE instructions. Each XMM register is 128 bits and is available for special purpose processing with the SSE instructions. Each of these XMM registers holds 16 bytes (8-bit), so four such registers can be used to store 64 bytes of data. Thus, by using SSE instructions (some of which work on different operand sizes, for example, treating each of the XMM registers as containing 16 one-byte operands), 64 bytes of data can be operated at a time using four consecutive SSE instructions (e.g., fetching from memory, storing into memory, zeroing, adding, multiplying), the remaining registers being used for intermediate results and temporary storage. With such an architecture, several routines are useful for optimizing the

28

byte-level performance, including the Parallel Lookup Multiplier, Sequencer 1, and Sequencer 2 discussed above.

While the above description contains many specific embodiments of the invention, these should not be construed as limitations on the scope of the invention, but rather as examples of specific embodiments thereof. Accordingly, the scope of the invention should be determined not by the embodiments illustrated, but by the appended claims and their equivalents.

GLOSSARY OF SOME VARIABLES

A encoding matrix ($F \times K$), sub-matrix of T

B encoding matrix ($F \times F$), sub-matrix of T

B^{-1} solution matrix ($F \times F$)

C encoded data matrix

$$((N + M) \times L) = \begin{bmatrix} D \\ J \end{bmatrix}$$

C' surviving encoded data matrix

$$(N \times L) = \begin{bmatrix} X \\ W \end{bmatrix}$$

D original data matrix ($N \times L$)

D' permuted original data matrix

$$(N \times L) = \begin{bmatrix} X \\ Y \end{bmatrix}$$

E information dispersal matrix

$$(IDM)((N + M) \times N) = \begin{bmatrix} I_N \\ H \end{bmatrix}$$

F number of failed data drives

G number of failed check drives

H check drive encoding matrix ($M \times N$)

I identity matrix ($I_K = K \times K$ identity matrix, $I_N = N \times N$ identity matrix)

J encoded check data matrix ($M \times L$)

K number of surviving data drives = $N - F$

L data block size (elements or bytes)

M number of check drives

M_{max} maximum value of M

N number of data drives

N_{max} maximum value of N

O zero matrix ($K \times F$), sub-matrix of T

S master encoding matrix ($(M_{max} + N_{max}) \times N_{max}$)

T transformed IDM

$$(N \times N) = \begin{bmatrix} I_K & O \\ A & B \end{bmatrix}$$

W surviving check data matrix ($F \times L$)

X surviving original data matrix ($K \times L$)

Y lost original data matrix ($F \times L$)

US 10,291,259 B2

29

What is claimed is:

1. A system adapted to use accelerated error-correcting code (ECC) processing to improve the storage and retrieval of digital data distributed across a plurality of drives, comprising:

at least one processor comprising at least one single-instruction-multiple-data (SIMD) central processing unit (CPU) core that executes SIMD instructions and loads original data from a main memory and stores check data to the main memory, the SIMD CPU core comprising at least 16 vector registers, each of the vector registers storing at least 16 bytes;

at least one system drive comprising at least one non-volatile storage medium that stores the SIMD instructions;

a plurality of data drives each comprising at least one non-volatile storage medium that stores at least one block of the original data, the at least one block comprising at least 512 bytes;

more than two check drives each comprising at least one non-volatile storage medium that stores at least one block of the check data;

at least one first input/output (I/O) controller that receives the at least one block of the original data from a transmitter and that stores the at least one block of the original data to the main memory; and

at least one second input/output (I/O) controller that stores the at least one block of the check data from the main memory to the check drives, wherein the processor, the SIMD instructions, the non-volatile storage medium, and the at least one second I/O controller are configured to implement an erasure coding system comprising:

a data matrix comprising at least one vector and comprising a plurality of rows of at least one block of the original data in the main memory, each of the rows being stored on a different one of the data drives;

a check matrix comprising more than two rows of the at least one block of the check data in the main memory, each of the rows being stored on a different one of the check drives, one of the rows comprising a parity row comprising the Galois Field (GF) summation of all of the rows of the data matrix; and

a thread that executes on the SIMD CPU core and comprising:

at least one parallel multiplier that multiplies the at least one vector of the data matrix by a single factor to compute parallel multiplier results comprising at least one vector;

at least one parallel adder that adds the at least one vector of the parallel multiplier results and computes a running total; and

a sequencer wherein the sequencer orders load operations of the original data into at least one of the vector registers and computes the check data with the parallel multiplier and the parallel adder, and stores the computed check data from the vector registers to the main memory.

2. The system of claim 1, wherein:

the processor comprises a first CPU core and a second CPU core;

the thread comprises a plurality of threads comprising a first thread group and a second thread group; and

the erasure coding system further comprises a scheduler for performing data operations to generate the check data and, concurrently, performing I/O operations using the at least one second I/O controller by:

30

assigning the data operations to the first thread group, and not assigning the I/O operations to the first thread group;

assigning the I/O operations to the second thread group and not assigning the data operations to the second thread group;

assigning the first thread group to the first CPU core; assigning the second thread group to the second CPU core; and

concurrently executing the first thread group on the first CPU core and the second thread group on the second CPU core to concurrently generate the check data and perform the I/O operations.

3. The system of claim 1, wherein the sequencer loads each entry of the data matrix from the main memory into a vector register at most once while generating the check data.

4. The system of claim 1, wherein the at least one processor is an x86 architecture processor.

5. The system of claim 1, wherein the erasure coding system further comprises:

an encoding matrix comprising more than two but not more than 254 rows and more than one but not more than 253 columns of factors in the main memory, wherein each of the entries of one of the rows of the encoding matrix comprises a multiplicative identity factor, the factors being for encoding the original data into the check data.

6. The system of claim 5, wherein the at least one parallel multiplier multiplies the at least one vector of the data matrix in units of at least 64 bytes.

7. The system of claim 5, wherein the data matrix comprises a first number of rows and the data drives comprise the first number of data drives,

wherein the check matrix comprises a second number of rows and the check drives comprise the second number of check drives, and

wherein the encoding matrix comprises a plurality of first factors in the second number of rows and the first number of columns.

8. The system of claim 7, wherein the encoding matrix further comprises a third number of columns and a plurality of second factors in the third number of columns,

wherein the data drives further comprise the third number of data drives, and

wherein the first factors are independent of the third number.

9. The system of claim 7, wherein the encoding matrix further comprises a fourth number of rows and a plurality of third factors in the fourth number of rows,

wherein the check drives further comprise the fourth number of check drives, and

wherein the first factors are independent of the fourth number.

10. The system of claim 5, wherein the multiplicative identity factor is 1.

11. The system of claim 5, wherein the at least one parallel multiplier multiplies the at least one vector of the data matrix by the single factor in the encoding matrix at a rate of less than about 2 machine instructions per byte of the data matrix.

12. A system adapted to use accelerated error-correcting code (ECC) processing to improve the storage and retrieval of digital data distributed across a plurality of drives, comprising:

at least one processor comprising at least one single-instruction-multiple-data (SIMD) central processing unit (CPU) core that executes SIMD instructions and

US 10,291,259 B2

31

loads surviving original data and surviving check data from a main memory and stores lost original data to the main memory, the SIMD CPU core comprising at least 16 vector registers, each of the vector registers storing at least 16 bytes;

at least one system drive comprising at least one non-volatile storage medium that stores the SIMD instructions;

a plurality of data drives each comprising at least one non-volatile storage medium that stores at least one block of the original data, the at least one block comprising at least 512 bytes;

more than two check drives each comprising at least one non-volatile storage medium that stores at least one block of the check data;

at least one first input/output (I/O) controller that transmits at least one block of computed lost original data from the main memory to a receiver; and

at least one second input/output (I/O) controller that reads at least one block of the check data from the check drives and stores the at least one block of the check data to the main memory,

wherein the processor, the SIMD instructions, the non-volatile storage medium and the at least one second I/O controller implement the accelerated ECC processing, comprising:

a surviving data matrix comprising at least one vector and comprising at least one row of at least one block of the surviving original data in the main memory, each row of the at least one row being stored on a different one of the data drives, and a lost data matrix comprising at least one block of the lost original data in the main memory;

a surviving check matrix comprising at least one row of at least one block of the surviving check data in the main memory, each row of the at least one row being stored on a different one of the check drives;

a solution matrix that holds factors in the main memory, the factors of the solution matrix being for decoding the surviving original data and the surviving check data into the lost original data; and

a thread that executes on the SIMD CPU core and comprising:

at least one parallel multiplier that multiplies the at least one vector of the surviving data matrix by a single factor in the solution matrix to compute parallel multiplier results comprising at least one vector;

at least one parallel adder that adds the at least one vector of the parallel multiplier results and computes a running total; and

a sequencer wherein the sequencer:

orders load operations of the surviving original data into at least one of the vector registers and load operations of the surviving check data into at least one of the vector registers;

computes the lost original data with the parallel multiplier and the parallel adder; and

stores the computed lost original data from the vector registers to the lost data matrix.

13. The system of claim 12, wherein:

the processing core comprises a first CPU core and a second CPU core;

the thread comprises a plurality of threads comprising a first thread group and a second thread group; and

the system further comprises a scheduler for performing data operations to regenerate the lost original data and,

32

concurrently, performing I/O operations using the at least one second I/O controller by:

assigning the data operations to the first thread group, and not assigning the I/O operations to the first thread group;

assigning the I/O operations to the second thread group, and not assigning the data operations to the second thread group;

assigning the first thread group to the first CPU core; assigning the second thread group to the second CPU core; and

concurrently executing the first thread group on the first CPU core and the second thread group on the second CPU core to concurrently regenerate the lost original data and perform the I/O operations.

14. The system of claim 12, wherein the sequencer loads each entry of the surviving original data from the main memory into a vector register at most once while regenerating the lost original data.

15. The system of claim 12, wherein the at least one parallel multiplier multiplies the at least one vector of the surviving data matrix in units of at least 64 bytes.

16. The system of claim 12, wherein the processor is an x86 architecture processor.

17. The system of claim 12, wherein the solution matrix comprises an inverted sub-matrix of an encoding matrix and wherein each of entries of one of the rows of the encoding matrix comprises a multiplicative identity factor, the factors of the encoding matrix being for encoding the original data into the check data.

18. The system of claim 17, wherein the multiplicative identity factor is 1.

19. The system of claim 12, wherein the at least one parallel multiplier multiplies the at least one vector of the surviving data matrix by the single factor in the solution matrix at a rate of less than about 2 machine instructions per byte of the surviving data matrix.

20. A method for accelerated error-correcting code (ECC) processing to improve the storage and retrieval of digital data distributed across a plurality of drives using a computing system, the computing system comprising:

at least one processor comprising at least one single-instruction-multiple-data (SIMD) central processing unit (CPU) core that executes a computer program including SIMD computer instructions and loads original data from a main memory and stores check data to the main memory, the SIMD CPU core comprising at least 16 vector registers, each of the vector registers storing at least 16 bytes;

at least one system drive comprising at least one non-volatile storage medium that stores the SIMD computer instructions;

a plurality of data drives each comprising at least one non-volatile storage medium that stores at least one block of the original data, the at least one block comprising at least 512 bytes;

more than two check drives each comprising at least one non-volatile storage medium that stores at least one block of the check data;

at least one first input/output (I/O) controller that receives the at least one block of the original data from a transmitter and that stores the at least one block of the original data to the main memory; and

at least one second input/output (I/O) controller that stores the at least one block of the check data from the main memory to the check drives, the method comprising:

accessing the SIMD instructions from the system drive;

US 10,291,259 B2

33

executing the SIMD instructions on the SIMD CPU core;

arranging the original data as a data matrix comprising at least one vector and comprising a plurality of rows of at least one block of the original data in the main memory, each of the rows being stored on a different one of the data drives;

arranging the check data as a check matrix comprising more than two rows of the at least one block of the check data in the main memory, each of the rows being stored on a different one of the check drives, one of the rows comprising a parity row comprising the Galois Field (GF) summation of all of the rows of the data matrix; and

encoding the original data into the check data using:

- at least one parallel multiplier that multiplies the at least one vector of the data matrix by a single factor to compute parallel multiplier results comprising at least one vector; and
- at least one parallel adder that adds the at least one vector of the parallel multiplier results and computes a running total,

the encoding of the check data comprising:

- loading the original data into at least one of the vector registers;
- computing the check data with the parallel multiplier and the parallel adder; and
- storing the computed check data from the vector registers into the main memory.

21. The method of claim 20, wherein:

- the processor comprises a first CPU core and a second CPU core;
- the executing of the SIMD instructions comprises executing the SIMD instructions on the first CPU core to perform data operations to generate the check data and, concurrently, to perform I/O operations on the second CPU core to control the at least one second I/O controller;
- the method further comprises scheduling the data operations concurrently with the I/O operations by:
 - assigning the data operations to the first CPU core, and not assigning the I/O operations to the first CPU core; and
 - assigning the I/O operations to the second CPU core and not assigning the data operations to the second CPU core.

22. The method of claim 20, further comprising loading each entry of the data matrix from the main memory into a vector register at most once while generating the check data.

23. The method of claim 20, wherein the processor is an x86 architecture processor.

24. The method of claim 20, further comprising:

- arranging factors as an encoding matrix comprising more than two but not more than 254 rows and more than one but not more than 253 columns of factors in the main memory, wherein each of the entries of one of the rows of the encoding matrix comprises a multiplicative identity factor, the factors being for encoding the original data into the check data.

25. The method of claim 24, wherein the at least one parallel multiplier multiplies the at least one vector of the data matrix in units of at least 64 bytes.

26. The method of claim 24, wherein the data matrix comprises a first number of rows and the data drives comprise the first number of data drives,

34

wherein the check matrix comprises a second number of rows and the check drives comprise the second number of check drives, and

wherein the encoding matrix comprises a plurality of first factors in the second number of rows and the first number of columns.

27. The method of claim 26, further comprising:

- adding a third number of data drives to the data drives by expanding the encoding matrix to further comprise the third number of columns and a plurality of second factors in the third number of columns,
- wherein the first factors are independent of the third number.

28. The method of claim 26, further comprising:

- adding a fourth number of check drives to the check drives by expanding the encoding matrix to further comprise the fourth number of rows and a plurality of third factors in the fourth number of rows,
- wherein the first factors are independent of the fourth number.

29. The method of claim 24, wherein the at least one parallel multiplier multiplies the at least one vector of the data matrix by the single factor in the encoding matrix at a rate of less than about 2 machine instructions per byte of the data matrix.

30. The method of claim 24, wherein the multiplicative identity factor is 1.

31. A method for accelerated error-correcting code (ECC) processing to improve the storage and retrieval of digital data distributed across a plurality of drives using a computing system, the computing system comprising:

- at least one processor comprising at least one single-instruction-multiple-data (SIMD) central processing unit (CPU) core that executes a computer program including SIMD instructions and loads surviving original data and surviving check data from a main memory and stores lost original data to the main memory, the SIMD CPU core comprising at least 16 vector registers, each of the vector registers storing at least 16 bytes;
- at least one system drive comprising at least one non-volatile storage medium that stores the SIMD instructions;
- a plurality of data drives each comprising at least one non-volatile storage medium that stores at least one block of the original data, the at least one block comprising at least 512 bytes;
- more than two check drives each comprising at least one non-volatile storage medium that stores at least one block of the check data;
- at least one first input/output (I/O) controller that transmit at least one block of computed lost original data from the main memory to a receiver; and
- at least one second input/output (I/O) controller that reads at least one block of the surviving check data from the check drives and stores the at least one block of the surviving check data to the main memory, the method comprising:
 - accessing the SIMD instructions from the system drive;
 - executing the SIMD instructions on the SIMD CPU core;
 - arranging the original data as a surviving data matrix comprising at least one vector and comprising at least one row of at least one block of the surviving original data in the main memory, each row of the at least one row being stored on a different one of the

US 10,291,259 B2

35

data drives, and a lost data matrix comprising at least one block of the lost original data in the main memory;

arranging factors as a solution matrix that holds the factors in the main memory, the factors being for decoding the surviving original data and the surviving check data into the lost original data, the surviving check data being arranged as a surviving check matrix comprising at least one row of at least one block of the surviving check data in the main memory, each row of the at least one row being stored on a different one of the check drives;

decoding the surviving check data into the lost original data using:

- at least one parallel multiplier that multiplies the at least one vector of the surviving data matrix by a single factor in the solution matrix to compute parallel multiplier results comprising at least one vector; and
- at least one parallel adder that adds the at least one vector of the parallel multiplier results and computes a running total,

the decoding the surviving check data into the lost original data comprising:

- loading the surviving original data into at least one of the vector registers;
- loading the surviving check data into at least one of the vector registers;
- computing the lost original data with the parallel multiplier and the parallel adder; and
- storing the computed lost original data from the vector registers into the lost data matrix.

32. The method of claim **31**, wherein:

- the processor comprises a first CPU core and a second CPU core;
- the executing of the SIMD instructions comprises executing the SIMD instructions on the first CPU core to perform data operations to reconstruct the lost original data and, concurrently, to perform I/O operations on the second CPU core to control the at least one second I/O controller;
- the method further comprises scheduling the data operations to be performed concurrently with the I/O operations by:
 - assigning the data operations to the first CPU core, and not assigning the I/O operations to the first CPU core; and
 - assigning the I/O operations to the second CPU core, and not assigning the data operations to the first CPU core.

33. The method of claim **31**, further comprising loading each entry of the surviving original data from the main memory into a vector register at most once while regenerating the lost original data.

34. The method of claim **31**, wherein the at least one parallel multiplier multiplies the at least one vector of the surviving data matrix in units of at least 64 bytes.

35. The method of claim **31**, wherein the processor is an x86 architecture processor.

36. The method of claim **31**, wherein the solution matrix comprises an inverted sub-matrix of an encoding matrix and wherein each of entries of one of the rows of the encoding matrix comprises a multiplicative identity factor, the factors of the encoding matrix being for encoding the original data into the check data.

37. The method of claim **36**, wherein the multiplicative identity factor is 1.

36

38. The method of claim **31**, wherein the at least one parallel multiplier multiplies the at least one vector of the surviving data matrix by the single factor in the solution matrix at a rate of less than about 2 machine instructions per byte of the surviving data matrix.

39. A system drive comprising at least one non-transitory computer-readable storage medium containing a computer program comprising a plurality of computer instructions that, when executed by a computing system, cause the computing system to perform accelerated error-correcting code (ECC) processing that improves the storage and retrieval of digital data distributed across a plurality of drives, the computing system comprising:

- at least one processor comprising at least one single-instruction-multiple-data (SIMD) central processing unit (CPU) core that executes SIMD instructions and loads original data from a main memory and stores check data to the main memory, the SIMD CPU core comprising at least 16 vector registers, each of the vector registers storing at least 16 bytes;
- a plurality of data drives each comprising at least one non-volatile storage medium that stores at least one block of the original data, the at least one block comprising at least 512 bytes;
- more than two check drives each comprising at least one non-volatile storage medium that stores at least one block of the check data;
- at least one first input/output (I/O) controller that receives the at least one block of the original data from a transmitter and that stores the at least one block of the original data to the main memory; and
- at least one second input/output (I/O) controller that stores the at least one block of the check data from the main memory to the check drives,

the computer instructions implementing protection of the original data in the main memory when executed on the computing system by:

- arranging the original data as a data matrix comprising at least one vector and comprising a plurality of rows of at least one block of the original data in the main memory, each of the rows being stored on a different one of the data drives;
- arranging the check data as a check matrix comprising more than two rows of the at least one block of the check data in the main memory, each of the rows being stored on a different one of the check drives, one of the rows comprising a parity row comprising the Galois Field (GF) summation of all of the rows of the data matrix; and
- encoding the original data into the check data using:
 - at least one parallel multiplier that multiplies the at least one vector of the data matrix by a single factor in an encoding matrix to compute parallel multiplier results comprising at least one vector; and
 - at least one parallel adder that adds the at least one vector of the parallel multiplier results and computes a running total,

the encoding the original data into the check data comprising:

- loading the original data into at least one of the vector registers;
- computing the check data with the parallel multiplier and the parallel adder; and
- storing the computed check data from the vector registers into the main memory.

US 10,291,259 B2

37

40. The system drive of claim 39, wherein: the processor comprises a first CPU core and a second CPU core;

the executing of the computer instructions comprises executing the computer instructions on the first CPU core to perform data operations to generate the check data and, concurrently, to perform I/O operations on the second CPU core to control the at least one second I/O controller;

the computer instructions implementing the protection of the original data comprise instructions that schedule the data operations to be performed concurrently with the I/O operations by:

assigning the data operations to the first CPU core, and not assigning the I/O operations to the first CPU core; and

assigning the I/O operations to the second CPU core and not assigning the data operations to the second CPU core.

41. The system drive of claim 39, wherein the computer instructions further comprise computer instructions that, when executed by the computing system, cause the computing system to load each entry of the data matrix from the main memory into a vector register at most once while generating the check data.

42. The system drive of claim 39, wherein the processor is an x86 architecture processor.

43. The system drive of claim 39, wherein the computer instructions implementing the protection of the original data comprise instructions to:

arrange factors as an encoding matrix comprising more than two but not more than 254 rows and more than one but not more than 253 columns of factors in the main memory, wherein each of the entries of one of the rows of the encoding matrix comprises a multiplicative identity factor, the factors being for encoding the original data into the check data.

44. The system drive of claim 43, wherein the at least one parallel multiplier multiplies the at least one vector of the data matrix in units of at least 64 bytes.

45. The system drive of claim 43, wherein the data matrix comprises a first number of rows and the data drives comprise the first number of data drives,

wherein the check matrix comprises a second number of rows and the check drives comprise the second number of check drives, and

wherein the encoding matrix comprises a plurality of first factors in the second number of rows and the first number of columns.

46. The system drive of claim 45, wherein the computer instructions further comprise instructions that, when executed on the computing system, cause the computing system to:

add a third number of data drives to the data drives by expanding the encoding matrix to further comprise the third number of columns and a plurality of second factors in the third number of columns,

wherein the first factors are independent of the third number.

47. The system drive of claim 45, wherein the computer instructions further comprise instructions that, when executed on the computing system, cause the computing system to:

add a fourth number of check drives to the check drives by expanding the encoding matrix to further comprise the fourth number of rows and a plurality of third factors in the fourth number of rows,

38

wherein the first factors are independent of the fourth number.

48. The system drive of claim 45, wherein the multiplicative identity factor is 1.

49. The system drive of claim 45, wherein the at least one parallel multiplier multiplies the at least one vector of the data matrix by the single factor in the encoding matrix at a rate of less than about 2 machine instructions per byte of the data matrix.

50. A system drive comprising at least one non-transitory computer-readable storage medium containing a computer program comprising a plurality of computer instructions that, when executed by a computing system, cause the computing system to perform accelerated error-correcting code (ECC) processing that improves the storage and retrieval of digital data distributed across a plurality of drives, the computing system comprising:

at least one processor comprising at least one single-instruction-multiple-data (SIMD) central processing unit (CPU) core that executes SIMD instructions and loads surviving original data and surviving check data from a main memory and stores lost original data to the main memory, the SIMD CPU core comprising at least 16 vector registers, each of the vector registers storing at least 16 bytes;

a plurality of data drives each comprising at least one non-volatile storage medium that stores at least one block of the original data, the at least one block comprising at least 512 bytes;

more than two check drives each comprising at least one non-volatile storage medium that stores at least one block of the check data;

at least one first input/output (I/O) controller that transmits at least one block of computed lost original data from the main memory to a receiver; and

at least one second input/output (I/O) controller that reads at least one block of the check data from the check drives and stores the at least one block of the check data to the main memory;

the computer instructions implementing protection of the original data in the main memory when executed on the computing system by:

arranging the surviving original data as a surviving data matrix comprising at least one vector and comprising at least one row of at least one block of the surviving original data in the main memory, each row of the at least one row being stored on a different one of the data drives, and a lost data matrix comprising at least one block of the lost original data in the main memory;

arranging factors as a solution matrix that holds the factors in the main memory, the factors being for decoding the surviving original data and the surviving check data into the lost original data, the surviving check data arranged as a surviving check matrix comprising at least one row of at least one block of the surviving check data in the main memory, each row of the at least one row being stored on a different one of the check drives; and

decoding the surviving check data into the lost original data using:

at least one parallel multiplier that multiplies the at least one vector of the surviving data matrix by a single factor in the solution matrix to compute parallel multiplier results comprising at least one vector; and

US 10,291,259 B2

39

at least one parallel adder that adds the at least one vector of the parallel multiplier results and computes a running total,

the decoding the surviving check data into the lost original data comprising:

- loading the surviving original data into at least one of the vector registers;
- loading the surviving check data into at least one of the vector registers;
- computing the lost original data with the parallel multiplier and the parallel adder; and
- storing the computed lost original data from the vector registers into the lost data matrix.

51. The system drive of claim 50, wherein:

the processor comprises a first CPU core and a second CPU core;

the executing of the computer instructions comprises executing the computer instructions on the first CPU core to perform data operations to reconstruct the lost original data and, concurrently, to perform I/O operations on the second CPU core to control the at least one second I/O controller;

the computer instructions further comprise instructions that schedule the data operations to be performed concurrently with the I/O operations by:

- assigning the data operations to the first CPU core, and not assigning the I/O operations to the first CPU core; and

40

assigning the I/O operations to the second CPU core, and not assigning the data operations to the first CPU core.

52. The system drive of claim 50, wherein the computer instructions further comprise computer instructions that, when executed on the computing system, cause the computing system to load each entry of the surviving original data from the main memory into a vector register at most once while regenerating the lost original data.

53. The system drive of claim 50, wherein the at least one parallel multiplier multiplies the at least one vector of the surviving data matrix in units of at least 64 bytes.

54. The system drive of claim 50, wherein the processor is an x86 architecture processor.

55. The system drive of claim 50 wherein the solution matrix comprises an inverted sub-matrix of an encoding matrix and wherein each of entries of one of the rows of the encoding matrix comprises a multiplicative identity factor, the factors of the encoding matrix being for encoding the original data into the check data.

56. The system drive of claim 55, wherein the multiplicative identity factor is 1.

57. The system drive of claim 50, wherein the at least one parallel multiplier multiplies the at least one vector of the surviving data matrix by the single factor in the solution matrix at a rate of less than about 2 machine instructions per byte of the surviving data matrix.

* * * * *

EXHIBIT G

PATENT

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

I hereby certify that this correspondence is being EFS-Web transmitted to the United States Patent and Trademark Office on February 23, 2021 at or before 11:59 p.m. Pacific Time under the Rules of 37 CFR § 1.8.

/Jennifer Guerra/
Jennifer Guerra

Inventor(s) : Michael H. Anderson et al. Confirmation No. 5095
Assignee : Streamscale, Inc.
Patent No. : 10,291,259
Issued : May 14, 2019
Application No. : 15/976,175
Filed : May 10, 2018
Title : ACCELERATED ERASURE CODING SYSTEM AND METHOD
Docket No. : 157162/411563-00014

**PETITION FOR CORRECTION OF INVENTORSHIP
UNDER 37 CFR § 1.324**

Mail Stop Petition
Commissioner for Patents
P.O. Box 1450
Alexandria, VA 22313-1450

Post Office Box 29001
Glendale, CA 91209-9001
February 23, 2021

Commissioner:

Pursuant to 37 C.F.R. §1.324, Applicant respectfully requests the correction of inventorship for the above issued patent to include inventor Sarah Mann. Ms. Mann was not named as an inventor through error.

Enclosed are:

(1) Statement of Sarah Mann in Support of Petition for Correction of Inventorship Pursuant to 37 C.F.R. §1.324;

(2) Statement of Michael Anderson in Support of Petition for Correction of Inventorship Pursuant to 37 C.F.R. §1.324;

Patent No. 10,291,259

(3) Statement of Assignee, Streamscale, Inc., in Support of Petition for Correction of Inventorship Pursuant to 37 C.F.R. §1.324 and Complying with 37 C.F.R. §3.73(c).

(4) Executed Inventors Declaration and Assignment document signed by Sarah Mann; and

(5) Application Data Sheet.

The required fee of \$160.00 as required by §1.20(b). The Commissioner is hereby authorized to charge any fees as required by this petition to Deposit Account No. 03-1728. Please show our docket number with any charge or credit to our deposit account.

Respectfully submitted,

LEWIS ROCA ROTHGERBER CHRISTIE LLP

By /David A. Plumley/

David A. Plumley

Reg. No. 37,208

626/795-9900

DAP/jhg
Enclosures

Under the Paperwork Reduction Act of 1995, no persons are required to respond to a collection of information unless it contains a valid OMB control number.

Application Data Sheet 37 CFR 1.76		Attorney Docket Number	157162/411563-00014
		Application Number	15976175
Title of Invention	ACCELERATED ERASURE CODING SYSTEM AND METHOD		
<p>The application data sheet is part of the provisional or nonprovisional application for which it is being submitted. The following form contains the bibliographic data arranged in a format specified by the United States Patent and Trademark Office as outlined in 37 CFR 1.76. This document may be completed electronically and submitted to the Office in electronic format using the Electronic Filing System (EFS) or the document may be printed and included in a paper filed application.</p>			

Secrecy Order 37 CFR 5.2:

Portions or all of the application associated with this Application Data Sheet may fall under a Secrecy Order pursuant to 37 CFR 5.2 (Paper filers only. Applications that fall under Secrecy Order may not be filed electronically.)

Inventor Information:

Inventor 1 Remove				
Legal Name				
Prefix	Given Name	Middle Name	Family Name	Suffix
	Michael	H.	Anderson	
Residence Information (Select One) <input checked="" type="radio"/> US Residency <input type="radio"/> Non US Residency <input type="radio"/> Active US Military Service				
City	Los Angeles	State/Province	CA	Country of Residence US

Mailing Address of Inventor:

Address 1	6423 Monterey Road, Unit 2			
Address 2				
City	Los Angeles	State/Province	CA	
Postal Code	90042	Country	US	

Inventor 2 Remove				
Legal Name				
Prefix	Given Name	Middle Name	Family Name	Suffix
	Sarah		Mann	
Residence Information (Select One) <input checked="" type="radio"/> US Residency <input type="radio"/> Non US Residency <input type="radio"/> Active US Military Service				
City	Oakland	State/Province	CA	Country of Residence US

Mailing Address of Inventor:

Address 1	196 Ridgeway Avenue			
Address 2				
City	Oakland	State/Province	CA	
Postal Code	94611	Country	US	

All Inventors Must Be Listed - Additional inventor information blocks may be generated within this form by selecting the Add button.

Add

Correspondence Information:

Enter either Customer Number or complete the Correspondence Information section below. For further information see 37 CFR 1.33(a).

Under the Paperwork Reduction Act of 1995, no persons are required to respond to a collection of information unless it contains a valid OMB control number.

Application Data Sheet 37 CFR 1.76		Attorney Docket Number	157162/411563-00014
		Application Number	
Title of Invention	ACCELERATED ERASURE CODING SYSTEM AND METHOD		

 An Address is being provided for the correspondence information of this application.

Customer Number	23363		
Email Address	PTO@LRRC.COM	<input type="button" value="Add Email"/>	<input type="button" value="Remove Email"/>

Application Information:

Title of the Invention	ACCELERATED ERASURE CODING SYSTEM AND METHOD		
Attorney Docket Number	157162/411563-00014	Small Entity Status Claimed	<input checked="" type="checkbox"/>
Application Type	Nonprovisional		
Subject Matter	Utility		
Total Number of Drawing Sheets (if any)	9	Suggested Figure for Publication (if any)	

Filing By Reference:

Only complete this section when filing an application by reference under 35 U.S.C. 111(c) and 37 CFR 1.57(a). Do not complete this section if application papers including a specification and any drawings are being filed. Any domestic benefit or foreign priority information must be provided in the appropriate section(s) below (i.e., "Domestic Benefit/National Stage Information" and "Foreign Priority Information").

For the purposes of a filing date under 37 CFR 1.53(b), the description and any drawings of the present application are replaced by this reference to the previously filed application, subject to conditions and requirements of 37 CFR 1.57(a).

Application number of the previously filed application	Filing date (YYYY-MM-DD)	Intellectual Property Authority or Country

Publication Information:
 Request Early Publication (Fee required at time of Request 37 CFR 1.219)

Request Not to Publish. I hereby request that the attached application not be published under 35 U.S.C. 122(b) and certify that the invention disclosed in the attached application **has not and will not** be the subject of an application filed in another country, or under a multilateral international agreement, that requires publication at eighteen months after filing.

Representative Information:

Representative information should be provided for all practitioners having a power of attorney in the application. Providing this information in the Application Data Sheet does not constitute a power of attorney in the application (see 37 CFR 1.32). Either enter Customer Number or complete the Representative Name section below. If both sections are completed the customer number will be used for the Representative Information during processing.

Please Select One:	<input checked="" type="radio"/> Customer Number	<input type="radio"/> US Patent Practitioner	<input type="radio"/> Limited Recognition (37 CFR 11.9)
Customer Number	23363		

Under the Paperwork Reduction Act of 1995, no persons are required to respond to a collection of information unless it contains a valid OMB control number.

Application Data Sheet 37 CFR 1.76		Attorney Docket Number	157162/411563-00014
		Application Number	
Title of Invention	ACCELERATED ERASURE CODING SYSTEM AND METHOD		

Domestic Benefit/National Stage Information:

This section allows for the applicant to either claim benefit under 35 U.S.C. 119(e), 120, 121, 365(c), or 386(c) or indicate National Stage entry from a PCT application. Providing benefit claim information in the Application Data Sheet constitutes the specific reference required by 35 U.S.C. 119(e) or 120, and 37 CFR 1.76.

When referring to the current application, please leave the "Application Number" field blank.

Prior Application Status	Pending		Remove		
Application Number	Continuity Type	Prior Application Number	Filing or 371(c) Date (YYYY-MM-DD)		
15976175	Continuation of	15201196	2016-07-01		
Prior Application Status	Patented		Remove		
Application Number	Continuity Type	Prior Application Number	Filing Date (YYYY-MM-DD)	Patent Number	Issue Date (YYYY-MM-DD)
15201196	Continuation of	14852438	2015-09-11	9385759	2016-07-05
Prior Application Status	Patented		Remove		
Application Number	Continuity Type	Prior Application Number	Filing Date (YYYY-MM-DD)	Patent Number	Issue Date (YYYY-MM-DD)
14852438	Continuation of	14223740	2014-03-24	9160374	2015-10-13
Prior Application Status	Patented		Remove		
Application Number	Continuity Type	Prior Application Number	Filing Date (YYYY-MM-DD)	Patent Number	Issue Date (YYYY-MM-DD)
14223740	Continuation of	13341833	2011-12-30	8683296	2014-03-25

Additional Domestic Benefit/National Stage Data may be generated within this form by selecting the **Add** button.

Foreign Priority Information:

This section allows for the applicant to claim priority to a foreign application. Providing this information in the application data sheet constitutes the claim for priority as required by 35 U.S.C. 119(b) and 37 CFR 1.55. When priority is claimed to a foreign application that is eligible for retrieval under the priority document exchange program (PDX)¹ the information will be used by the Office to automatically attempt retrieval pursuant to 37 CFR 1.55(i)(1) and (2). Under the PDX program, applicant bears the ultimate responsibility for ensuring that a copy of the foreign application is received by the Office from the participating foreign intellectual property office, or a certified copy of the foreign priority application is filed, within the time period specified in 37 CFR 1.55(g)(1).

Remove			
Application Number	Country ¹	Filing Date (YYYY-MM-DD)	Access Code ¹ (if applicable)

Additional Foreign Priority Data may be generated within this form by selecting the **Add** button.

Under the Paperwork Reduction Act of 1995, no persons are required to respond to a collection of information unless it contains a valid OMB control number.

Application Data Sheet 37 CFR 1.76	Attorney Docket Number	157162/411563-00014
	Application Number	
Title of Invention	ACCELERATED ERASURE CODING SYSTEM AND METHOD	

Statement under 37 CFR 1.55 or 1.78 for AIA (First Inventor to File) Transition Applications

<input type="checkbox"/> This application (1) claims priority to or the benefit of an application filed before March 16, 2013 and (2) also contains, or contained at any time, a claim to a claimed invention that has an effective filing date on or after March 16, 2013. NOTE: By providing this statement under 37 CFR 1.55 or 1.78, this application, with a filing date on or after March 16, 2013, will be examined under the first inventor to file provisions of the AIA.

Application Data Sheet 37 CFR 1.76	Attorney Docket Number	157162/411563-00014
	Application Number	
Title of invention	ACCELERATED ERASURE CODING SYSTEM AND METHOD	

Authorization or Opt-Out of Authorization to Permit Access:

When this Application Data Sheet is properly signed and filed with the application, applicant has provided written authority to permit a participating foreign intellectual property (IP) office access to the instant application-as-filed (see paragraph A in subsection 1 below) and the European Patent Office (EPO) access to any search results from the instant application (see paragraph B in subsection 1 below).

Should applicant choose not to provide an authorization identified in subsection 1 below, applicant **must opt-out** of the authorization by checking the corresponding box A or B or both in subsection 2 below.

NOTE: This section of the Application Data Sheet is **ONLY** reviewed and processed with the **INITIAL** filing of an application. After the initial filing of an application, an Application Data Sheet cannot be used to provide or rescind authorization for access by a foreign IP office(s). Instead, Form PTO/SB/39 or PTO/SB/69 must be used as appropriate.

1. Authorization to Permit Access by a Foreign Intellectual Property Office(s)

A. Priority Document Exchange (PDX) - Unless box A in subsection 2 (opt-out of authorization) is checked, the undersigned hereby **grants the USPTO authority** to provide the European Patent Office (EPO), the Japan Patent Office (JPO), the Korean Intellectual Property Office (KIPO), the State Intellectual Property Office of the People's Republic of China (SIPO), the World Intellectual Property Organization (WIPO), and any other foreign intellectual property office participating with the USPTO in a bilateral or multilateral priority document exchange agreement in which a foreign application claiming priority to the instant patent application is filed, access to: (1) the instant patent application-as-filed and its related bibliographic data, (2) any foreign or domestic application to which priority or benefit is claimed by the instant application and its related bibliographic data, and (3) the date of filing of this Authorization. See 37 CFR 1.14(h)(1).

B. Search Results from U.S. Application to EPO - Unless box B in subsection 2 (opt-out of authorization) is checked, the undersigned hereby **grants the USPTO authority** to provide the EPO access to the bibliographic data and search results from the instant patent application when a European patent application claiming priority to the instant patent application is filed. See 37 CFR 1.14(h)(2).

The applicant is reminded that the EPO's Rule 141(1) EPC (European Patent Convention) requires applicants to submit a copy of search results from the instant application without delay in a European patent application that claims priority to the instant application.

2. Opt-Out of Authorizations to Permit Access by a Foreign Intellectual Property Office(s)

A. Applicant **DOES NOT** authorize the USPTO to permit a participating foreign IP office access to the instant application-as-filed. If this box is checked, the USPTO will not be providing a participating foreign IP office with any documents and information identified in subsection 1A above.

B. Applicant **DOES NOT** authorize the USPTO to transmit to the EPO any search results from the instant patent application. If this box is checked, the USPTO will not be providing the EPO with search results from the instant application.

NOTE: Once the application has published or is otherwise publicly available, the USPTO may provide access to the application in accordance with 37 CFR 1.14.

Under the Paperwork Reduction Act of 1995, no persons are required to respond to a collection of information unless it contains a valid OMB control number.

Application Data Sheet 37 CFR 1.76	Attorney Docket Number	157162/411563-00014
	Application Number	
Title of invention	ACCELERATED ERASURE CODING SYSTEM AND METHOD	

Applicant Information:

Providing assignment information in this section does not substitute for compliance with any requirement of part 3 of Title 37 of CFR to have an assignment recorded by the Office

Applicant 1

If the applicant is the inventor (or the remaining joint inventor or inventors under 37 CFR 1.45), this section should not be completed. The information to be provided in this section is the name and address of the legal representative who is the applicant under 37 CFR 1.43; or the name and address of the assignee, person to whom the inventor is under an obligation to assign the invention, or person who otherwise shows sufficient proprietary interest in the matter who is the applicant under 37 CFR 1.46. If the applicant is an applicant under 37 CFR 1.46 (assignee, person to whom the inventor is obligated to assign, or person who otherwise shows sufficient proprietary interest) together with one or more joint inventors, then the joint inventor or inventors who are also the applicant should be identified in this section.

Assignee
 Legal Representative under 35 U.S.C. 117
 Joint Inventor
 Person to whom the inventor is obligated to assign.
 Person who shows sufficient proprietary interest

If applicant is the legal representative, indicate the authority to file the patent application, the inventor is:

Name of the Deceased or Legally Incapacitated Inventor:

If the Applicant is an Organization check here.

Organization Name:

Mailing Address Information For Applicant:

Address 1	8423 Monterey Road, Unit 2 7215 Bosque Blvd., Suite 203		
Address 2			
City	Los Angeles <u>Waco</u>	State/Province	CA <u>TX</u>
Country	US	Postal Code	90042 <u>76710</u>
Phone Number		Fax Number	
Email Address			

Additional Applicant Data may be generated within this form by selecting the Add button.

Assignee Information including Non-Applicant Assignee Information:

Providing assignment information in this section does not substitute for compliance with any requirement of part 3 of Title 37 of CFR to have an assignment recorded by the Office.

Under the Paperwork Reduction Act of 1995, no persons are required to respond to a collection of information unless it contains a valid OMB control number.

Application Data Sheet 37 CFR 1.76		Attorney Docket Number	157162/411563-00014
		Application Number	
Title of Invention	ACCELERATED ERASURE CODING SYSTEM AND METHOD		

Assignee 1

Complete this section if assignee information, including non-applicant assignee information, is desired to be included on the patent application publication. An assignee-applicant identified in the "Applicant Information" section will appear on the patent application publication as an applicant. For an assignee-applicant, complete this section only if identification as an assignee is also desired on the patent application publication.

If the Assignee or Non-Applicant Assignee is an Organization check here.

Prefix	Given Name	Middle Name	Family Name	Suffix

Mailing Address Information For Assignee including Non-Applicant Assignee:

Address 1				
Address 2				
City		State/Province		
Country ¹	Postal Code			
Phone Number		Fax Number		
Email Address				

Additional Assignee or Non-Applicant Assignee Data may be generated within this form by selecting the Add button.

Signature:

NOTE: This Application Data Sheet must be signed in accordance with 37 CFR 1.33(b). However, if this Application Data Sheet is submitted with the INITIAL filing of the application and either box A or B is not checked in subsection 2 of the "Authorization or Opt-Out of Authorization to Permit Access" section, then this form must also be signed in accordance with 37 CFR 1.14(c).

This Application Data Sheet **must** be signed by a patent practitioner if one or more of the applicants is a juristic entity (e.g., corporation or association). If the applicant is two or more joint inventors, this form must be signed by a patent practitioner, **all** joint inventors who are the applicant, or one or more joint inventor-applicants who have been given power of attorney (e.g., see USPTO Form PTO/AIA/81) on behalf of **all** joint inventor-applicants.

See 37 CFR 1.4(d) for the manner of making signatures and certifications.

Signature	/David A. Plumley/		Date (YYYY-MM-DD)	2021-02-23
First Name	David A.	Last Name	Plumley	Registration Number
37208				

Additional Signature may be generated within this form by selecting the Add button.

Under the Paperwork Reduction Act of 1995, no persons are required to respond to a collection of information unless it contains a valid OMB control number.

Application Data Sheet 37 CFR 1.76	Attorney Docket Number	157162/411563-00014
	Application Number	
Title of invention	ACCELERATED ERASURE CODING SYSTEM AND METHOD	

This collection of information is required by 37 CFR 1.76. The information is required to obtain or retain a benefit by the public which is to file (and by the USPTO to process) an application. Confidentiality is governed by 35 U.S.C. 122 and 37 CFR 1.14. This collection is estimated to take 23 minutes to complete, including gathering, preparing, and submitting the completed application data sheet form to the USPTO. Time will vary depending upon the individual case. Any comments on the amount of time you require to complete this form and/or suggestions for reducing this burden, should be sent to the Chief Information Officer, U.S. Patent and Trademark Office, U.S. Department of Commerce, P.O. Box 1450, Alexandria, VA 22313-1450. **DO NOT SEND FEES OR COMPLETED FORMS TO THIS ADDRESS. SEND TO: Commissioner for Patents, P.O. Box 1450, Alexandria, VA 22313-1450.**

Privacy Act Statement

The Privacy Act of 1974 (P.L. 93-579) requires that you be given certain information in connection with your submission of the attached form related to a patent application or patent. Accordingly, pursuant to the requirements of the Act, please be advised that: (1) the general authority for the collection of this information is 35 U.S.C. 2(b)(2); (2) furnishing of the information solicited is voluntary; and (3) the principal purpose for which the information is used by the U.S. Patent and Trademark Office is to process and/or examine your submission related to a patent application or patent. If you do not furnish the requested information, the U.S. Patent and Trademark Office may not be able to process and/or examine your submission, which may result in termination of proceedings or abandonment of the application or expiration of the patent.

The information provided by you in this form will be subject to the following routine uses:

1. The information on this form will be treated confidentially to the extent allowed under the Freedom of Information Act (5 U.S.C. 552) and the Privacy Act (5 U.S.C. 552a). Records from this system of records may be disclosed to the Department of Justice to determine whether the Freedom of Information Act requires disclosure of these records.
2. A record from this system of records may be disclosed, as a routine use, in the course of presenting evidence to a court, magistrate, or administrative tribunal, including disclosures to opposing counsel in the course of settlement negotiations.
3. A record in this system of records may be disclosed, as a routine use, to a Member of Congress submitting a request involving an individual, to whom the record pertains, when the individual has requested assistance from the Member with respect to the subject matter of the record.
4. A record in this system of records may be disclosed, as a routine use, to a contractor of the Agency having need for the information in order to perform a contract. Recipients of information shall be required to comply with the requirements of the Privacy Act of 1974, as amended, pursuant to 5 U.S.C. 552a(m).
5. A record related to an International Application filed under the Patent Cooperation Treaty in this system of records may be disclosed, as a routine use, to the International Bureau of the World Intellectual Property Organization, pursuant to the Patent Cooperation Treaty.
6. A record in this system of records may be disclosed, as a routine use, to another federal agency for purposes of National Security review (35 U.S.C. 181) and for review pursuant to the Atomic Energy Act (42 U.S.C. 218(c)).
7. A record from this system of records may be disclosed, as a routine use, to the Administrator, General Services, or his/her designee, during an inspection of records conducted by GSA as part of that agency's responsibility to recommend improvements in records management practices and programs, under authority of 44 U.S.C. 2904 and 2906. Such disclosure shall be made in accordance with the GSA regulations governing inspection of records for this purpose, and any other relevant (i.e., GSA or Commerce) directive. Such disclosure shall not be used to make determinations about individuals.
8. A record from this system of records may be disclosed, as a routine use, to the public after either publication of the application pursuant to 35 U.S.C. 122(b) or issuance of a patent pursuant to 35 U.S.C. 151. Further, a record may be disclosed, subject to the limitations of 37 CFR 1.14, as a routine use, to the public if the record was filed in an application which became abandoned or in which the proceedings were terminated and which application is referenced by either a published application, an application open to public inspections or an issued patent.
9. A record from this system of records may be disclosed, as a routine use, to a Federal, State, or local law enforcement agency, if the USPTO becomes aware of a violation or potential violation of law or regulation.

PATENT

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

Inventor(s)	: Michael H. Anderson et al.	Confirmation No. 5095
Assignee	: STREAMSCALE, INC.	
Patent No.	: 10,291,259	
Issued	: May 14, 2019	
Application No.	: 15/976,175	
Filed	: May 10, 2018	
Title	: ACCELERATED ERASURE CODING SYSTEM AND METHOD	
Docket No.	: 157162 (411563-00014)	

STATEMENT OF MICHAEL H. ANDERSON IN SUPPORT OF PETITION FOR CORRECTION OF INVENTORSHIP PURSUANT TO 37 C.F.R. § 1.324

Commissioner for Patents
P.O. Box 1450
Alexandria, VA 22313-1450

Post Office Box 29001
Glendale, CA 91209-9001


Commissioner:

I, the undersigned, declare and state as follows:

I submit this statement in support of STREAMSCALE, INC.'s petition to correct the inventorship in the above-identified patent. I am the named inventor of the above-identified patent. I understand that the petition seeks to add SARAH MANN as an inventor to this patent and I agree to the requested change of inventorship.

Executed this 20 of Feb, 2021 in Udon Thani
Thailand

Respectfully,



Michael H. Anderson

DAP/srd

**INVENTOR'S DECLARATION AND ASSIGNMENT
FOR PATENT APPLICATION**

PATENT

Title of Invention: ACCELERATED ERASURE CODING SYSTEM AND METHOD

Docket No.: 157162 (411563-00014)

Application No. 15/976,175

INVENTOR'S DECLARATION

As a below named inventor, I hereby declare that:

This declaration is directed to the attached application unless the following is checked:

United States Application or PCT International Application Number 15/976,175 filed on May 10, 2018.

The above-identified application was made or authorized to be made by me.

I believe that I am the original inventor or an original joint inventor of a claimed invention in the above-identified application.

I have reviewed and understand the contents of the above-identified application, including the claims.

I acknowledge the duty to disclose information which is material to patentability as defined in 37 C.F.R. § 1.56, including for continuation-in-part applications, material information which became available between the filing date of the prior application and the national or PCT international filing date of the continuation-in-part application.

I acknowledge that any willful false statement made in this declaration is punishable under 18 U.S.C. § 1001 by fine or imprisonment of not more than five (5) years, or both.

ASSIGNMENT

In consideration of good and valuable consideration, the receipt of which is hereby acknowledged, the undersigned,

(1) Sarah Mann

HEREBY SELL(S), ASSIGN(S) AND TRANSFER(S) TO

(2) STREAMSCALE, INC.

having a place of business at

(3) 7215 Bosque Blvd., Suite 203, Waco, Texas 76710

(hereinafter called "ASSIGNEE") the entire right, title and interest in and to any and all improvements which are disclosed in the application for United States Letters Patent entitled

(4) ACCELERATED ERASURE CODING SYSTEM AND METHOD

which application was executed on even date herewith or was

**INVENTOR'S DECLARATION AND ASSIGNMENT
FOR PATENT APPLICATION**

Docket No.: 157162 (411563-00014)

Application No.: 15/976,175

(a) executed on (5a): _____;
(b) filed on (5b): May 10, 2018 _____;
Application No.: 15/976,175 _____;

(LEWIS ROCA ROTHGERBER CHRISTIE
LLP, P.O. Box 29001, Glendale, CA 91209-
9001) is hereby authorized to insert in (b) the
specified data, when known.

including any and all United States Patents
which may be granted on said application, and any and all extensions, divisions, reissues,
substitutes, renewals or continuations of said application and patents, and the right to all benefits
under all international conventions for the protection of industrial property and applications for
said improvements.

It is hereby authorized and requested that the Commissioner of Patents issue any and all of said
Letters Patent, when granted, to said ASSIGNEE, its assigns or its successors in interest or its
designee.

Upon said consideration, it is further agreed that, when requested, without charge to but at the
expense of said ASSIGNEE, the undersigned will execute all divisional, continuing, substitute,
renewal, and reissue patent applications; execute all rightful other papers; and generally do
everything possible which said ASSIGNEE shall consider desirable for aiding in securing and
maintaining patent protection as provided herein.

Sarah Mann
Legal Name of Inventor

2/18/2021
Date

DocuSigned by:
/Sarah Mann/
Signature

WITNESSES:

PATENT

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

Inventor(s)	: Michael H. Anderson et al.	Confirmation No. 5095
Assignee	: STREAMSCALE, INC.	
Patent No.	: 10,291,259	
Issued	: May 14, 2019	
Application No.	: 15/976,175	
Filed	: May 10, 2018	
Title	: ACCELERATED ERASURE CODING SYSTEM AND METHOD	
Docket No.	: 157162 (411563-00014)	

**STATEMENT OF ASSIGNEE IN SUPPORT OF PETITION
FOR CORRECTION OF INVENTORSHIP UNDER 37 C.F.R. § 1.324 AND
COMPLYING WITH 37 C.F.R. § 3.73(c)**

Commissioner for Patents
P.O. Box 1450
Alexandria, VA 22313-1450

Post Office Box 29001
Glendale, CA 91209-9001

Commissioner:

1. I am authorized to act on behalf of STREAMSCALE, INC. and have the title indicated below.

2. STREAMSCALE, INC. is the assignee of the entire interest of the patent identified above, by virtue of the following Assignments from the inventors.

(a) An Assignment of this invention by inventor Michael H. Anderson was recorded on May 16, 2018 at Reel No. 045816 and Frame No. 0289.

(b) A second Assignment of this invention by inventor Sarah Mann, the inventor to be added on this patent, is attached hereto.

3. The Assignee agrees to the addition of Sarah Mann as an inventor on the patent.

Date Feb 20, 2021

STREAMSCALE, INC.
 By: 
 Name: Michael H. Anderson
 Title: President

DAP/srd

PATENT

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

Inventor(s)	: Michael H. Anderson et al.	Confirmation No. 5095
Assignee	: STREAMSCALE, INC.	
Patent No.	: 10,291,259	
Issued	: May 14, 2019	
Application No.	: 15/976,175	
Filed	: May 10, 2018	
Title	: ACCELERATED ERASURE CODING SYSTEM AND METHOD	
Docket No.	: 157162 (411563-00014)	

STATEMENT OF SARAH MANN IN SUPPORT OF PETITION FOR CORRECTION OF INVENTORSHIP PURSUANT TO 37 C.F.R. § 1.324

Commissioner for Patents
P.O. Box 1450
Alexandria, VA 22313-1450

Post Office Box 29001
Glendale, CA 91209-9001


Commissioner:

I, the undersigned, declare and state as follows:

I submit this statement in support of STREAMSCALE, INC.'s petition to correct the inventorship in the above-identified patent. I understand that the petition seeks to add me, the undersigned, as an inventor to this patent and I agree to the requested change of inventorship.

Executed this 18 of February, 2021 in oakland,
CA

Respectfully,

DocuSigned by:


 Sarah Mann

DAP/srd

**INVENTOR'S DECLARATION AND ASSIGNMENT
FOR PATENT APPLICATION**

PATENT

Title of Invention: ACCELERATED ERASURE CODING SYSTEM AND METHOD

Docket No.: 157162 (411563-00014)

Application No. 15/976,175

INVENTOR'S DECLARATION

As a below named inventor, I hereby declare that:

This declaration is directed to the attached application unless the following is checked:

United States Application or PCT International Application Number 15/976,175 filed on May 10, 2018.

The above-identified application was made or authorized to be made by me.

I believe that I am the original inventor or an original joint inventor of a claimed invention in the above-identified application.

I have reviewed and understand the contents of the above-identified application, including the claims.

I acknowledge the duty to disclose information which is material to patentability as defined in 37 C.F.R. § 1.56, including for continuation-in-part applications, material information which became available between the filing date of the prior application and the national or PCT international filing date of the continuation-in-part application.

I acknowledge that any willful false statement made in this declaration is punishable under 18 U.S.C. § 1001 by fine or imprisonment of not more than five (5) years, or both.

ASSIGNMENT

In consideration of good and valuable consideration, the receipt of which is hereby acknowledged, the undersigned,

(1) Sarah Mann

HEREBY SELL(S), ASSIGN(S) AND TRANSFER(S) TO

(2) STREAMSCALE, INC.

having a place of business at

(3) 7215 Bosque Blvd., Suite 203, Waco, Texas 76710

(hereinafter called "ASSIGNEE") the entire right, title and interest in and to any and all improvements which are disclosed in the application for United States Letters Patent entitled

(4) ACCELERATED ERASURE CODING SYSTEM AND METHOD

which application was executed on even date herewith or was

**INVENTOR'S DECLARATION AND ASSIGNMENT
FOR PATENT APPLICATION**

Docket No.: 157162 (411563-00014)

Application No.: 15/976,175

(a) executed on (5a): _____;

(b) filed on (5b): May 10, 2018 _____;

Application No.: 15/976,175 _____;

(LEWIS ROCA ROTHGERBER CHRISTIE
LLP, P.O. Box 29001, Glendale, CA 91209-
9001) is hereby authorized to insert in (b) the
specified data, when known.

including any and all United States Patents
which may be granted on said application, and any and all extensions, divisions, reissues,
substitutes, renewals or continuations of said application and patents, and the right to all benefits
under all international conventions for the protection of industrial property and applications for
said improvements.

It is hereby authorized and requested that the Commissioner of Patents issue any and all of said
Letters Patent, when granted, to said ASSIGNEE, its assigns or its successors in interest or its
designee.

Upon said consideration, it is further agreed that, when requested, without charge to but at the
expense of said ASSIGNEE, the undersigned will execute all divisional, continuing, substitute,
renewal, and reissue patent applications; execute all rightful other papers; and generally do
everything possible which said ASSIGNEE shall consider desirable for aiding in securing and
maintaining patent protection as provided herein.

Sarah Mann
Legal Name of Inventor

2/18/2021
Date

DocuSigned by:
/Sarah Mann/
Signature

WITNESSES:

Electronic Patent Application Fee Transmittal

Application Number:	15976175			
Filing Date:	10-May-2018			
Title of Invention:	ACCELERATED ERASURE CODING SYSTEM AND METHOD			
First Named Inventor/Applicant Name:	Michael H. Anderson			
Filer:	David A. Plumley/Jennifer Guerra			
Attorney Docket Number:	157162/411563-00014			
Filed as Small Entity				
Filing Fees for Utility under 35 USC 111(a)				
Description	Fee Code	Quantity	Amount	Sub-Total in USD(\$)
Basic Filing:				
Pages:				
Claims:				
Miscellaneous-Filing:				
Petition:				
Patent-Appeals-and-Interference:				
Post-Allowance-and-Post-Issuance:				
PROCESSING FEE CORRECTING INVENTORSHIP	2816	1	160	160

Description	Fee Code	Quantity	Amount	Sub-Total in USD(\$)
Extension-of-Time:				
Miscellaneous:				
Total in USD (\$)				160

Electronic Acknowledgement Receipt

EFS ID:	42002889
Application Number:	15976175
International Application Number:	
Confirmation Number:	5095
Title of Invention:	ACCELERATED ERASURE CODING SYSTEM AND METHOD
First Named Inventor/Applicant Name:	Michael H. Anderson
Customer Number:	23363
Filer:	David A. Plumley/Jennifer Guerra
Filer Authorized By:	David A. Plumley
Attorney Docket Number:	157162/411563-00014
Receipt Date:	23-FEB-2021
Filing Date:	10-MAY-2018
Time Stamp:	20:54:53
Application Type:	Utility under 35 USC 111(a)

Payment information:

Submitted with Payment	yes
Payment Type	DA
Payment was successfully received in RAM	\$160
RAM confirmation Number	E20212MK55084580
Deposit Account	
Authorized User	

The Director of the USPTO is hereby authorized to charge indicated fees and credit any overpayment as follows:

File Listing:

Document Number	Document Description	File Name	File Size(Bytes)/ Message Digest	Multi Part /.zip	Pages (if appl.)
1	Petition for review by the Office of Petitions	157162_Petition.pdf	105663	no	2
			014454dd240ed03ac5e38018a2fada4a74e eebd7		

Warnings:**Information:**

2	Application Data Sheet	157162_CorrectedADS.pdf	4848887	no	9
			7a29237b634a1b2c0314d48f6a62f254895 92967		

Warnings:**Information:**

This is not an USPTO supplied ADS fillable form

3	Examination support document	157162_Stm_Anderson.pdf	294455	no	1
			a9bb66cd5190b5b1764424e475731af8c23 96551		

Warnings:**Information:**

4		157162_Stm_StreamScale.pdf	445215	yes	3
			1007b91ec7c027f6437850317936c2e3c98 16244		

Multipart Description/PDF files in .zip description

Document Description	Start	End
Assignee showing of ownership per 37 CFR 3.73	2	3
Examination support document	1	1

Warnings:**Information:**

5	Examination support document	157162_Stm_Mann.pdf	205328	no	1
			32fbc2e15f9d14666085c651447f5b99cfca ab60		

Warnings:

The PDF file has been signed with a digital signature and the legal effect of the document will be based on the contents of the file not the digital signature.

Information:

6	Oath or Declaration filed	157162_DeclAsg.pdf	211055	no	2
			40d7b794b6feb781e6f01f33c0fd402edac2085e		

Warnings:

The PDF file has been signed with a digital signature and the legal effect of the document will be based on the contents of the file not the digital signature.

Information:

7	Fee Worksheet (SB06)	fee-info.pdf	30496	no	2
			9335c3147ded48be7905137d4196ed5df12bbe5a		

Warnings:**Information:**

Total Files Size (in bytes):	6141099
-------------------------------------	---------

This Acknowledgement Receipt evidences receipt on the noted date by the USPTO of the indicated documents, characterized by the applicant, and including page counts, where applicable. It serves as evidence of receipt similar to a Post Card, as described in MPEP 503.

New Applications Under 35 U.S.C. 111

If a new application is being filed and the application includes the necessary components for a filing date (see 37 CFR 1.53(b)-(d) and MPEP 506), a Filing Receipt (37 CFR 1.54) will be issued in due course and the date shown on this Acknowledgement Receipt will establish the filing date of the application.

National Stage of an International Application under 35 U.S.C. 371

If a timely submission to enter the national stage of an international application is compliant with the conditions of 35 U.S.C. 371 and other applicable requirements a Form PCT/DO/EO/903 indicating acceptance of the application as a national stage submission under 35 U.S.C. 371 will be issued in addition to the Filing Receipt, in due course.

New International Application Filed with the USPTO as a Receiving Office

If a new international application is being filed and the international application includes the necessary components for an international filing date (see PCT Article 11 and MPEP 1810), a Notification of the International Application Number and of the International Filing Date (Form PCT/RO/105) will be issued in due course, subject to prescriptions concerning national security, and the date shown on this Acknowledgement Receipt will establish the international filing date of the application.

EXHIBIT H



US010666296B2

(12) **United States Patent**
Anderson

(10) **Patent No.:** **US 10,666,296 B2**
(45) **Date of Patent:** ***May 26, 2020**

(54) **ACCELERATED ERASURE CODING SYSTEM AND METHOD**

(58) **Field of Classification Search**
CPC G06F 11/1076; G06F 3/0619; G06F 3/064;
G06F 3/0683; G06F 11/1096;

(71) Applicant: **STREAMSCALE, INC.**, Los Angeles, CA (US)

(Continued)

(56) **References Cited**

(72) Inventor: **Michael H. Anderson**, Los Angeles, CA (US)

U.S. PATENT DOCUMENTS

(73) Assignee: **STREAMSCALE, INC.**, Los Angeles, CA (US)

5,577,054 A 11/1996 Pharris
5,754,563 A 5/1998 White
(Continued)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 0 days.

OTHER PUBLICATIONS

This patent is subject to a terminal disclaimer.

Casey Henderson: Letter to the USENIX Community <https://www.usenix.org/system/files/conference/fast13/fast13_memo_021715.pdf> Feb. 17, 2015.

(Continued)

(21) Appl. No.: **16/358,602**

Primary Examiner — John J Tabone, Jr.

(22) Filed: **Mar. 19, 2019**

(74) *Attorney, Agent, or Firm* — Lewis Roca Rothgerber Christie LLP

(65) **Prior Publication Data**

US 2019/0215013 A1 Jul. 11, 2019

(57) **ABSTRACT**

An accelerated erasure coding system includes a processing core for executing computer instructions and accessing data from a main memory, and a non-volatile storage medium for storing the computer instructions. The processing core, storage medium, and computer instructions are configured to implement an erasure coding system, which includes: a data matrix for holding original data in the main memory; a check matrix for holding check data in the main memory; an encoding matrix for holding first factors in the main memory, the first factors being for encoding the original data into the check data; and a thread for executing on the processing core. The thread includes: a parallel multiplier for concurrently multiplying multiple entries of the data matrix by a single entry of the encoding matrix; and a first sequencer for ordering operations through the data matrix and the encoding matrix using the parallel multiplier to generate the check data.

Related U.S. Application Data

(63) Continuation of application No. 15/976,175, filed on May 10, 2018, now Pat. No. 10,291,259, which is a (Continued)

(51) **Int. Cl.**

H03M 13/15 (2006.01)

G06F 11/10 (2006.01)

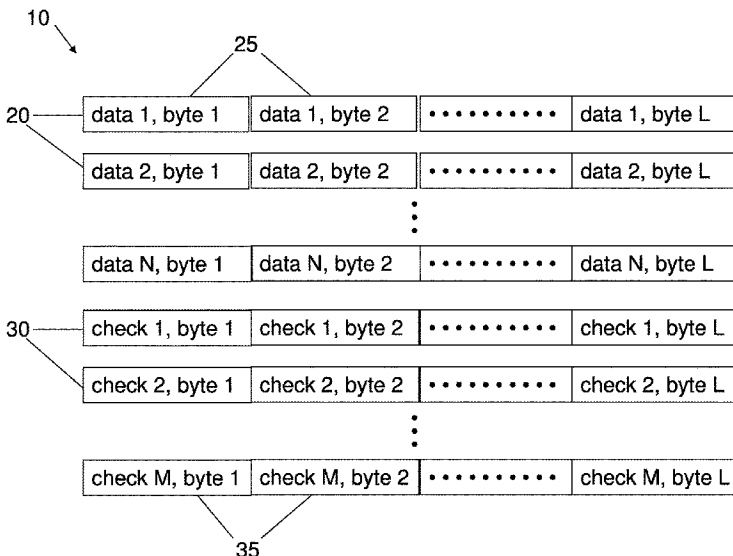
(Continued)

(52) **U.S. Cl.**

CPC **H03M 13/154** (2013.01); **G06F 11/1068** (2013.01); **G06F 11/1076** (2013.01);

(Continued)

8 Claims, 9 Drawing Sheets



US 10,666,296 B2

Page 2

Related U.S. Application Data

continuation of application No. 15/201,196, filed on Jul. 1, 2016, now Pat. No. 10,003,358, which is a continuation of application No. 14/852,438, filed on Sep. 11, 2015, now Pat. No. 9,385,759, which is a continuation of application No. 14/223,740, filed on Mar. 24, 2014, now Pat. No. 9,160,374, which is a continuation of application No. 13/341,833, filed on Dec. 30, 2011, now Pat. No. 8,683,296.

(51) Int. Cl.

H03M 13/11 (2006.01)
H03M 13/13 (2006.01)
G06F 12/02 (2006.01)
G06F 12/06 (2006.01)
H03M 13/37 (2006.01)
H04L 1/00 (2006.01)
H03M 13/00 (2006.01)
G11C 29/52 (2006.01)

(52) U.S. Cl.

CPC **G06F 11/1092** (2013.01); **G06F 11/1096** (2013.01); **G06F 12/0238** (2013.01); **G06F 12/06** (2013.01); **G11C 29/52** (2013.01); **H03M 13/1191** (2013.01); **H03M 13/134** (2013.01); **H03M 13/1515** (2013.01); **H03M 13/373** (2013.01); **H03M 13/3761** (2013.01); **H03M 13/3776** (2013.01); **H03M 13/616** (2013.01); **H03M 13/6502** (2013.01); **H04L 1/0043** (2013.01); **H04L 1/0057** (2013.01); **G06F 2211/109** (2013.01); **G06F 2211/1057** (2013.01)

(58) Field of Classification Search

CPC .. G06F 12/0238; G06F 12/06; G06F 11/1092; G06F 2211/1057; G06F 2211/109; H03M 13/11; H03M 13/1191; H03M 13/134; H03M 13/1515; H03M 13/154; H03M 13/158; H03M 13/373; H03M 13/3761; H03M 13/3776; H03M 13/616; H04L 1/0043
 USPC 714/764, 6.24, 6.1, 6.11, 6.2, 6.21, 6.32, 714/763, 752, 758, 768, 770, 773, 784, 714/786

See application file for complete search history.

(56)

References Cited

U.S. PATENT DOCUMENTS

6,486,803 B1 11/2002 Luby et al.
 6,654,924 B1* 11/2003 Hassner G11B 20/1813
 714/758
 6,823,425 B2* 11/2004 Ghosh G06F 11/1076
 711/114
 7,350,126 B2* 3/2008 Winograd G06F 11/1076
 714/752
 7,865,809 B1 1/2011 Lee et al.
 7,930,337 B2 4/2011 Hasenplaugh et al.
 8,145,941 B2* 3/2012 Jacobson G06F 11/1076
 714/6.24
 8,352,847 B2* 1/2013 Gunnam G06F 17/16
 714/758
 8,683,296 B2* 3/2014 Anderson H03M 13/1515
 714/763
 8,914,706 B2* 12/2014 Anderson G06F 11/1076
 714/6.24
 9,160,374 B2 10/2015 Anderson
 9,258,014 B2 2/2016 Anderson
 9,385,759 B2 7/2016 Anderson

10,003,358 B2 6/2018 Anderson
 2009/0055717 A1 2/2009 Au et al.
 2009/0249170 A1 10/2009 Maiuzzo
 2010/0293439 A1 11/2010 Flynn et al.
 2011/0029756 A1* 2/2011 Biscondi H03M 13/1114
 712/22
 2012/0272036 A1* 10/2012 Muralimanohar .. G06F 12/0238
 711/202
 2013/0108048 A1* 5/2013 Grube H04W 12/00
 380/270
 2013/0110962 A1* 5/2013 Grube H04W 12/00
 709/213
 2013/0111552 A1* 5/2013 Grube H04W 12/00
 726/3
 2013/0124932 A1* 5/2013 Schuh G11C 29/16
 714/718
 2013/0173956 A1* 7/2013 Anderson G06F 11/1076
 714/6.24
 2013/0173996 A1* 7/2013 Anderson H03M 13/3761
 714/770
 2014/0040708 A1 2/2014 Maiuzzo
 2014/0068391 A1 3/2014 Goel et al.
 2015/0012796 A1* 1/2015 Anderson H03M 13/3761
 714/763
 2017/0005671 A1 1/2017 Anderson

OTHER PUBLICATIONS

Chandan Kumar Singh: EC Jerasure plugin and StreamScale Inc, <<http://www.spinics.net/lists/ceph-devel/msg29944.html>> Apr. 20, 2016.

Code Poetry and Text Adventures: <<http://catid.mechafetus.com/news/news.php?view=381>> Dec. 14, 2014.

Curtis Chan: StreamScale Announces Settlement of Erasure Code Technology Patent Litigation, <<http://www.prweb.com/releases/2014/12/prweb12368357.htm>>, Dec. 3, 2014.

Ethan Miller, <<https://plus.google.com/113956021908222328905/posts/bPcYevPkJWd>>, Aug. 2, 2013.

H. Peter Anvin. "The mathematics of RAID-6." 2004, 2011.

Hafner et al., Matrix Methods for Lost Data Reconstruction in Erasure Codes, Nov. 16, 2005, USENIX FAST '05 Paper, pp. 1-26.

James S. Plank, Ethan L. Miller, Will B. Houston: GF-Complete: A Comprehensive Open Source Library for Galois Field Arithmetic, <<http://web.eecs.utk.edu/~plank/plank/papers/CS-13-703.html>> Jan. 2013.

James S. Plank, Jianqiang Luo, Catherine D. Schuman, Lihao Xu, Zooko Wilcox-O'Hearn: A Performance Evaluation and Examination of Open-Source Erasure Coding Libraries For Storage, <https://www.usenix.org/legacy/event/fast09/tech/full_papers/plank/plank.html> 2009.

Kevin M. Greenan, Ethan L. Miller, Thomas J.E. Schwarz, S. J.: Optimizing Galois Field Arithmetic for Diverse Processor Architectures and Applications, *Proceedings of the 16th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS 2008)*, Baltimore, MD, Sep. 2008.

Lee, "High-Speed VLSI Architecture for Parallel Reed-Solomon Decoder", IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 11, No. 2, Apr. 2003, pp. 288-294.

Li et al.; Preventing Silent Data Corruptions from Propagating During Data Reconstruction; IEEE Transactions on Computers, vol. 59, No. 12, Dec. 2010; pp. 1611-1624.

Li Han and Qian Huan-yan. "Parallelized Network Coding with SIMD instruction sets." In *Computer Science and Computational Technology, 2008. ISCSCT'08. International Symposium on*, vol. 1, pp. 364-369. IEEE, 2008.

Loic Dachary: Deadline of Github pull request for Hammer release, <<http://www.spinics.net/lists/ceph-devel/msg22001.html>> Jan. 13, 2015.

Louis Lavile: <<https://twitter.com/louislavile>> Nov. 13, 2014.

M. Lalam, et al. "Sliding Encoding-Window for Reed-Solomon code decoding," 4th International Symposium on Turbo Codes &

US 10,666,296 B2

Page 3

(56)

References Cited

OTHER PUBLICATIONS

Related Topics; 6th International ITG-Conference on Source and Channel Coding, Munich, Germany, 2006, pp. 1-6.

Maddock, et al.; White Paper, Surviving Two Disk Failures Introducing Various “RAID 6” Implementations; Xyratex; pp. 1-13.

Mann, “*The Original View of Reed-Solomon Coding and the Welch-Berlekamp Decoding Algorithm*”, A Dissertation Submitted to the Faculty of the Graduate Interdisciplinary Program in Applied Mathematics, The University of Arizona, Jul. 19, 2013, 143 sheets.

Marius Gedminas: <<http://eavesdrop.openstack.org/irclogs/%23openstack-swift/%23openstack-swift.2015-04-30.log.html>> Apr. 29, 2015.

Matthew L. Curry, Anthony Skjellum, H. Lee Ward, and Ron Brightwell. “Arbitrary dimension reed-solomon coding and decoding for extended raid on gpus.” In *Petascale Data Storage Workshop, 2008. PDSW’08. 3rd*, pp. 1-3. IEEE, 2008.

Matthew L. Curry, Anthony Skjellum, H. Lee Ward, Ron Brightwell: Gibraltar: A Reed-Solomon coding library for storage applications on programmable graphics processors. *Concurrency and Computation: Practice and Experience* 23(18): pp. 2477-2495 (2011).

Matthew L. Curry, H. Lee Ward, Anthony Skjellum, Ron Brightwell: A Lightweight, GPU-Based Software RAID System. *ICPP 2010*: pp. 565-572.

Matthew L. Curry, Lee H. Ward, Anthony Skjellum, and Ron B. Brightwell: Accelerating Reed-Solomon Coding in RAID Systems With GPUs, *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*. IEEE, 2008.

Michael A. O’Shea: StreamScale, <<https://lists.ubuntu.com/archives/technical-board/2015-April/002100.html>> Apr. 29, 2015.

Mike Masnik: Patent Troll Kills Open Source Project on Speeding Up The Computation of Erasure Codes, <<https://www.techdirt.com/articles/20141115/07113529155/patent-troll-kills-open-source-project-speeding-up-computation-erasure-codes.shtml>>, Nov. 19, 2014.

Neifeld, M.A & Sridharan, S. K. (1997). Parallel error correction using spectral Reed-Solomon codes. *Journal of Optical Communications*, 18(4), pp. 144-150.

Plank; All About Erasure Codes:—Reed-Solomon Coding—LDPC Coding; Logistical Computing and Internetworking Laboratory, Department of Computer Science, University of Tennessee; ICL—Aug. 20, 2004; 52 sheets.

Robert Louis Cloud, Matthew L. Curry, H. Lee Ward, Anthony Skjellum, Purushotham Bangalore: Accelerating Lossless Data Compression with GPUs. *CoRR abs/1107.1525* (2011).

Roy Schestowitz: US Patent Reform (on Trolls Only) More or Less Buried or Ineffective, <<http://techrighs.org/2014/12/12/us-patent-reform/>> Dec. 12, 2014.

Weibin Sun, Robert Ricci, Matthew L. Curry: GPUstore: harnessing GPU computing for storage systems in the OS kernel. *SYSTOR 2012*: p. 6.

Xin Zhou, Anthony Skjellum, Matthew L. Curry: Abstract: Extended Abstract for Evaluating Asynchrony in Gibraltar RAID’s GPU Reed-Solomon Coding Library. *SC Companion 2012*: pp. 1496-1497.

Xin Zhou, Anthony Skjellum, Matthew L. Curry: Poster: Evaluating Asynchrony in Gibraltar RAID’s GPU Reed-Solomon Coding Library. *SC Companion 2012*: p. 1498.

* cited by examiner

FIG. 1

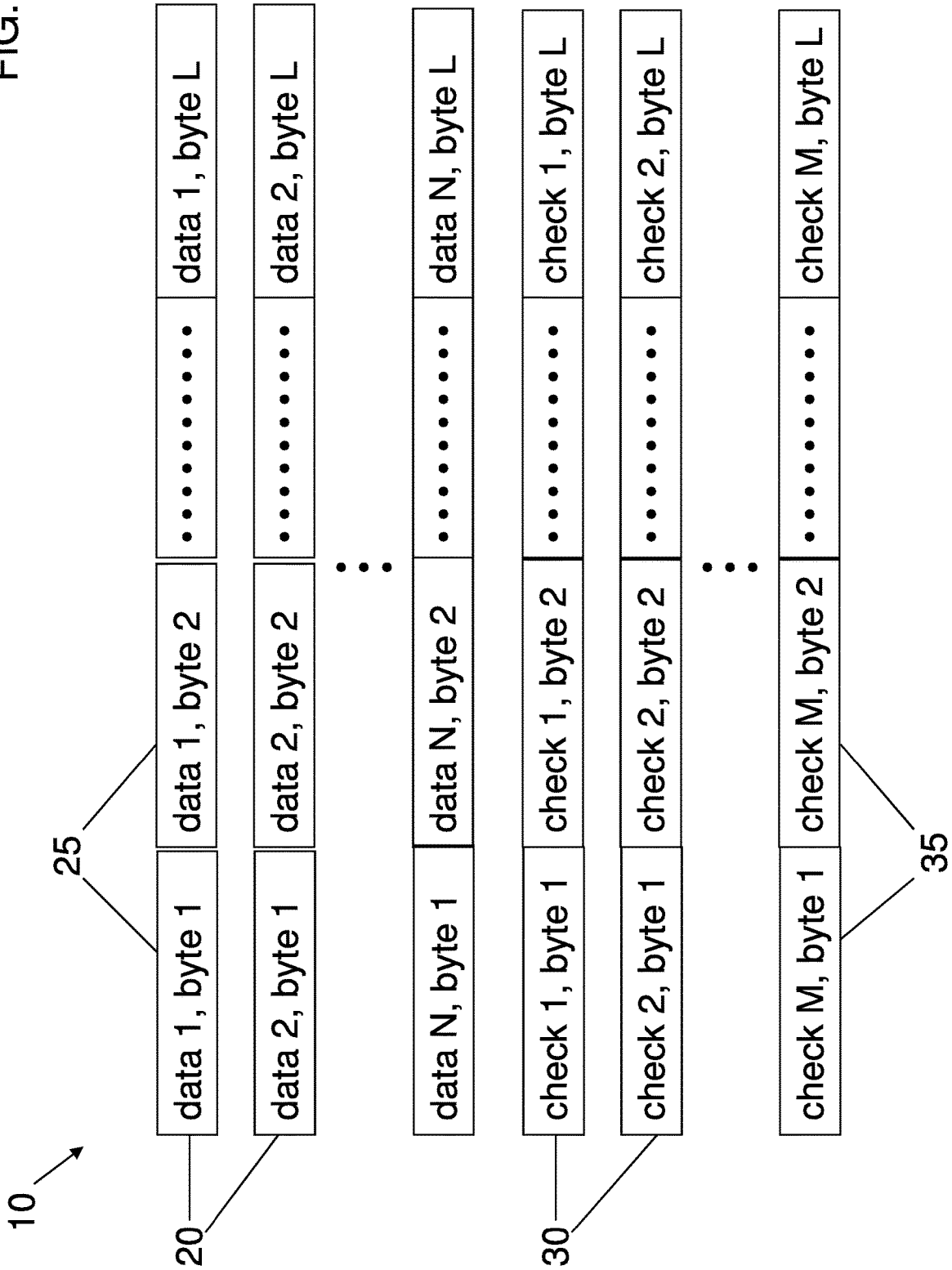


FIG. 2

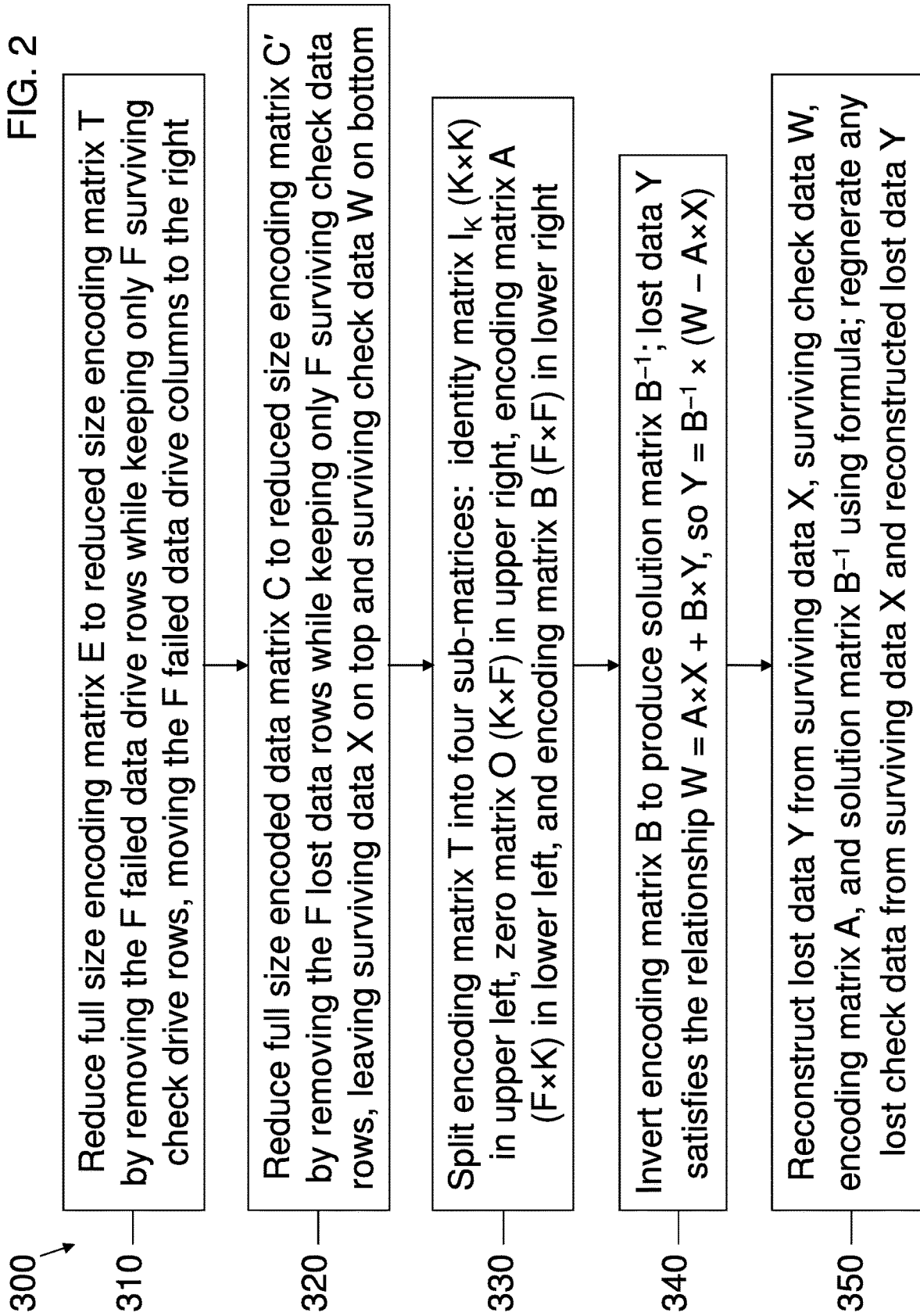


FIG. 3

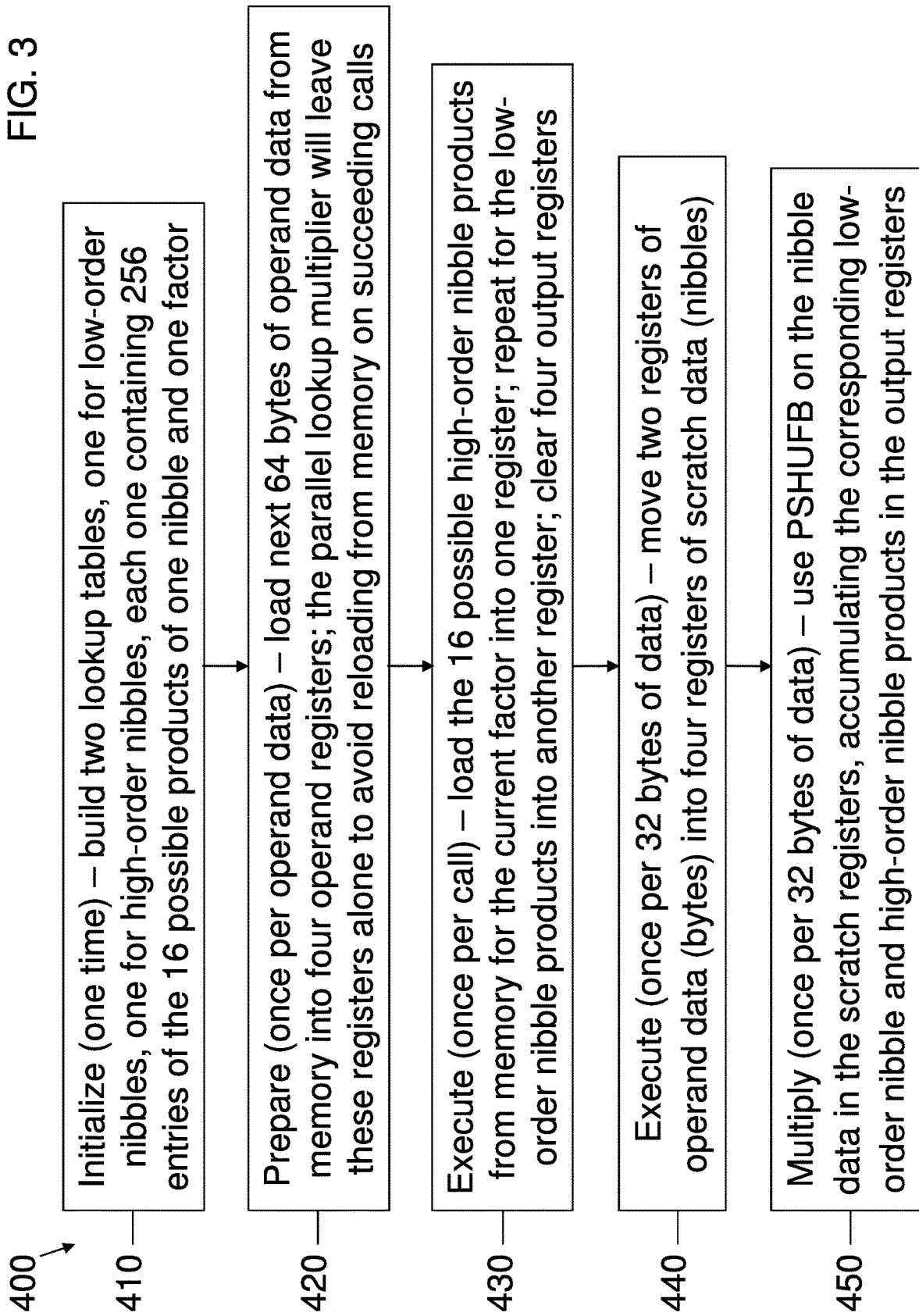


FIG. 4

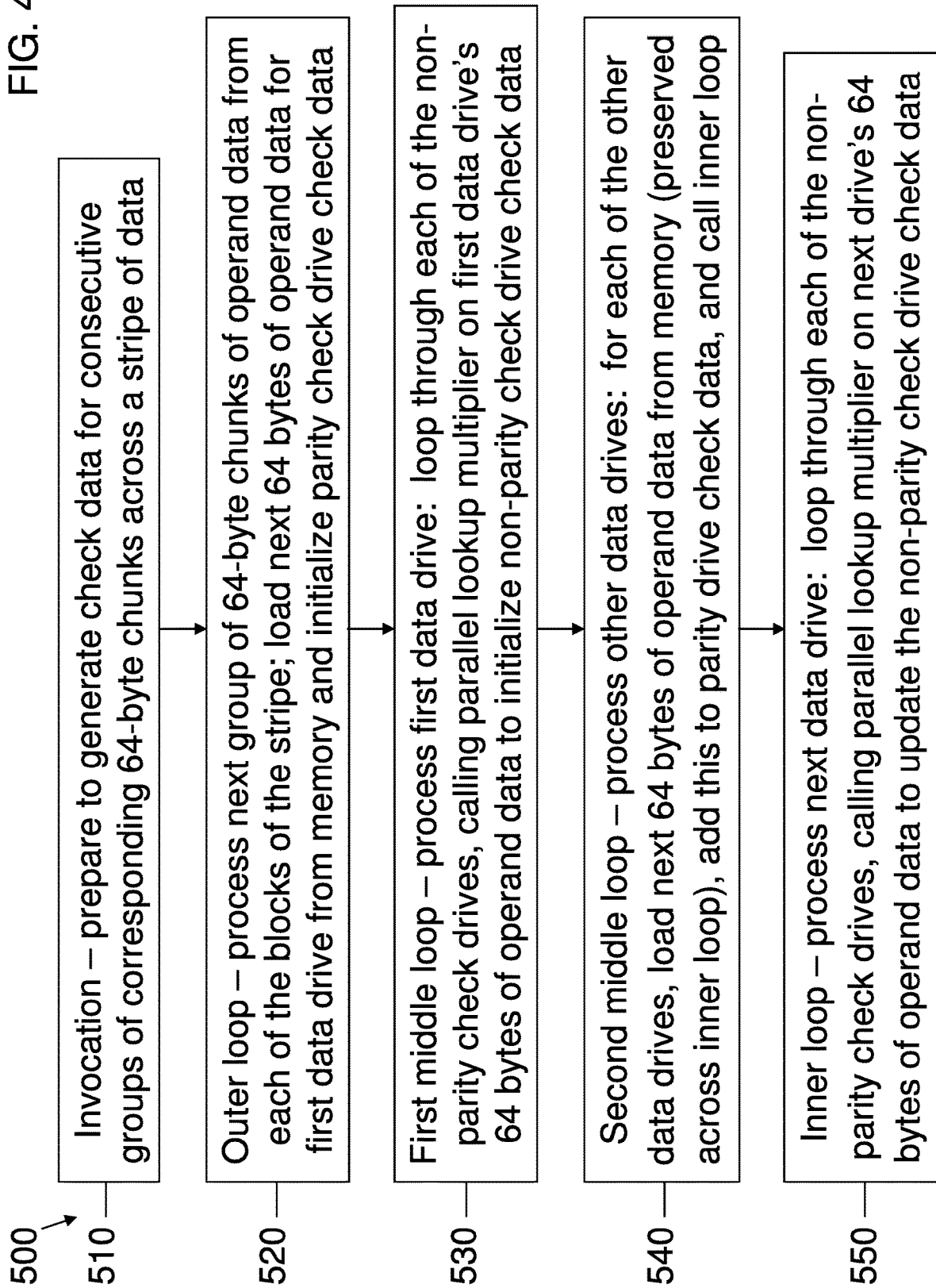


FIG. 5

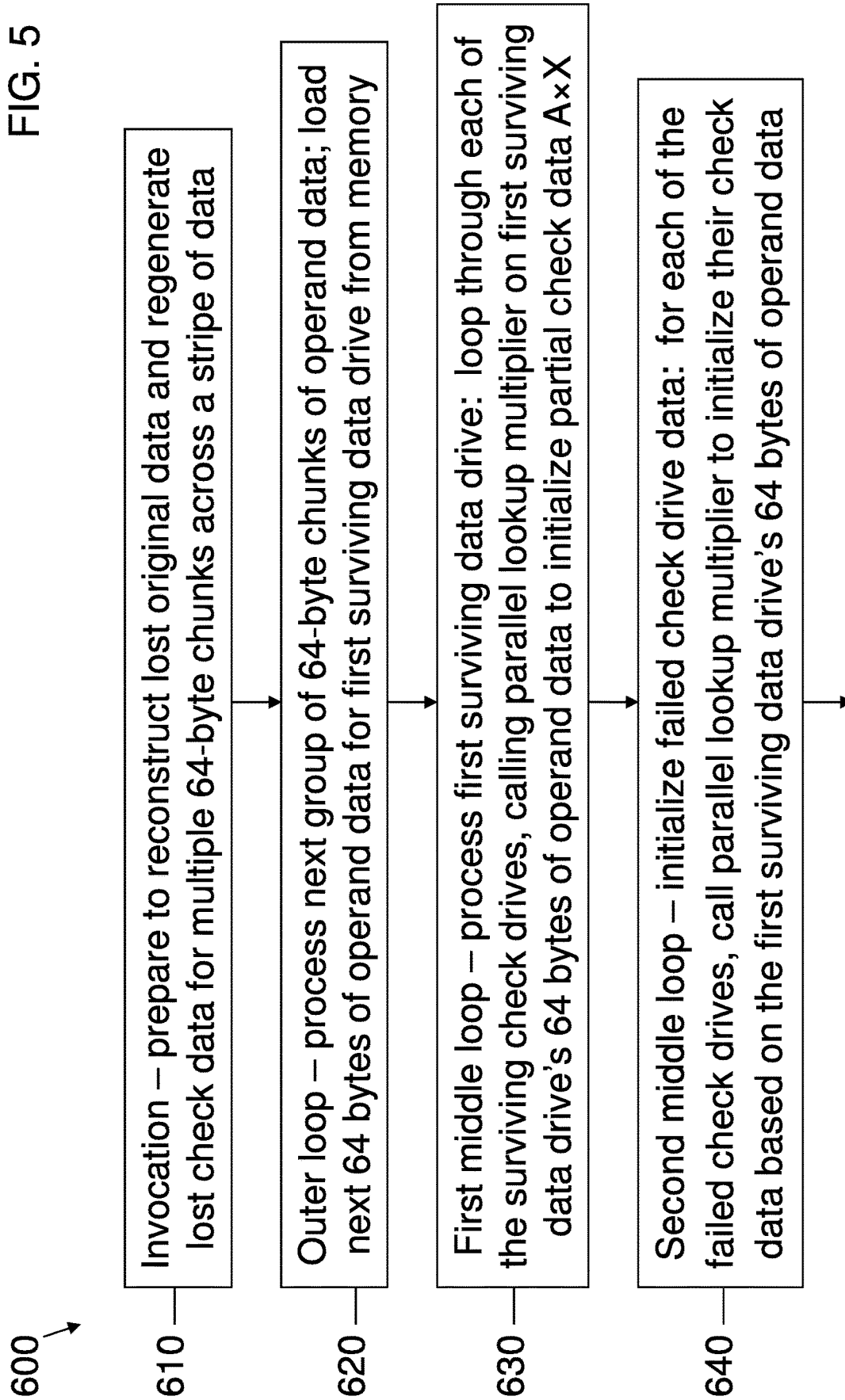


FIG. 6

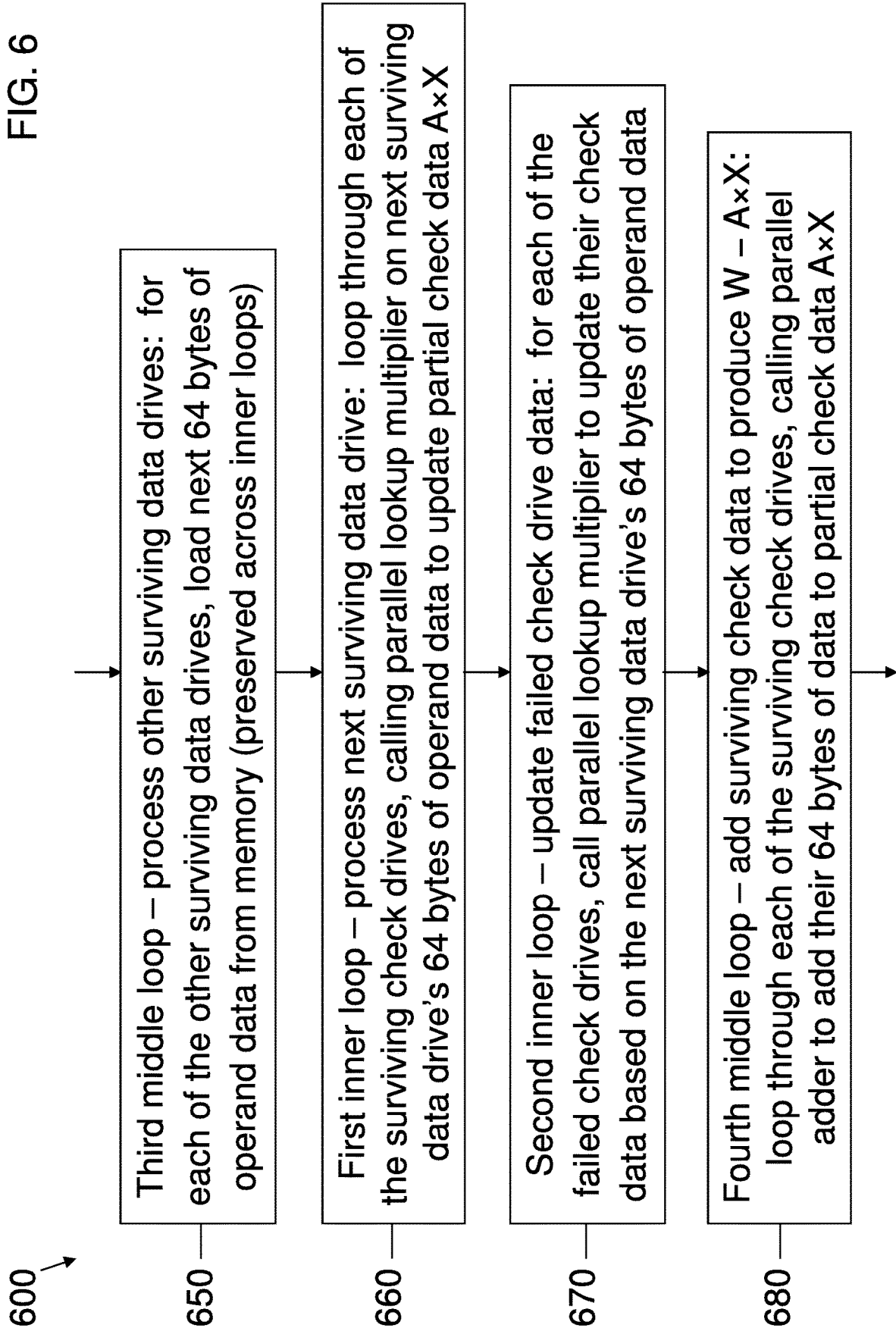


FIG. 7

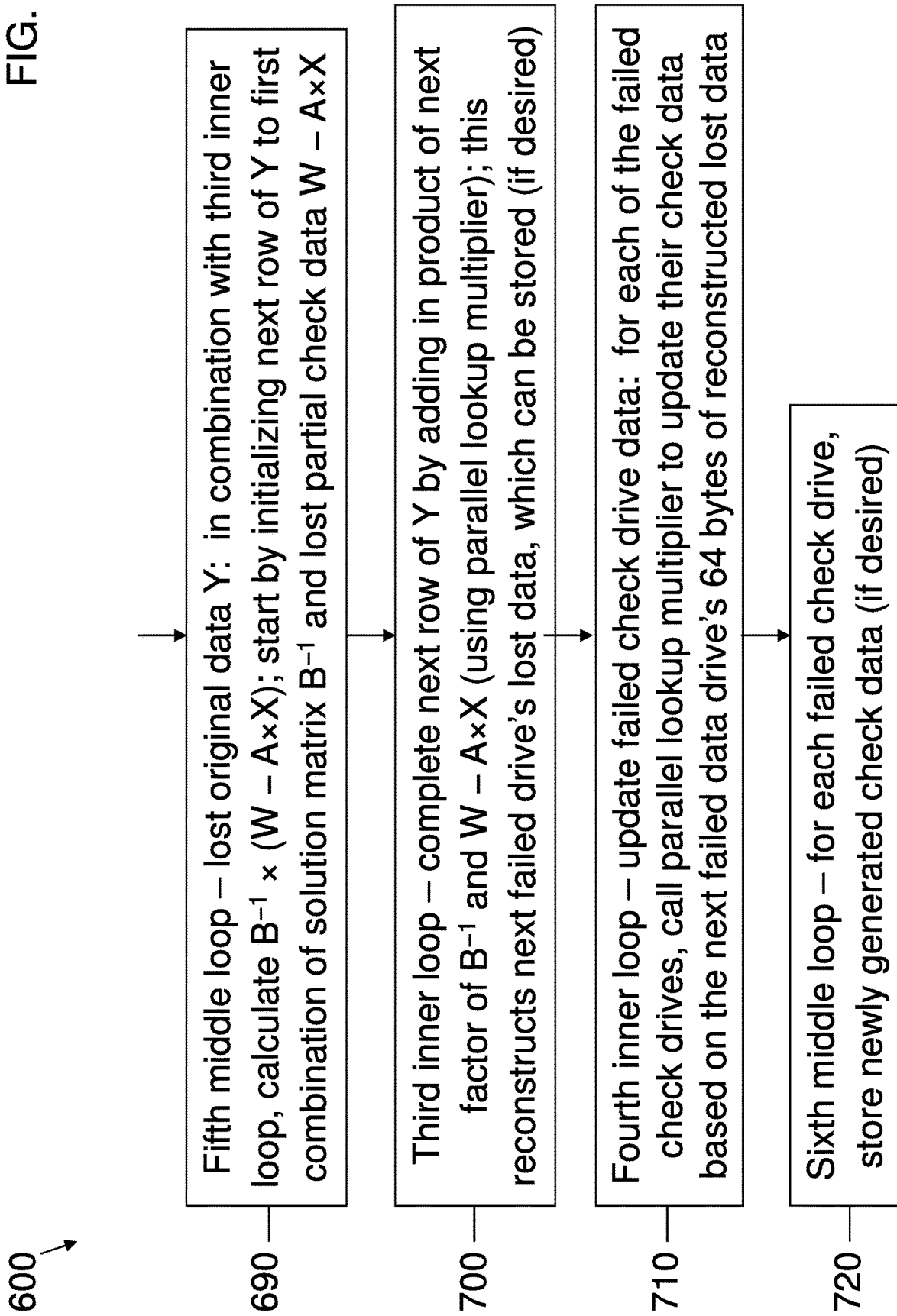


FIG. 8

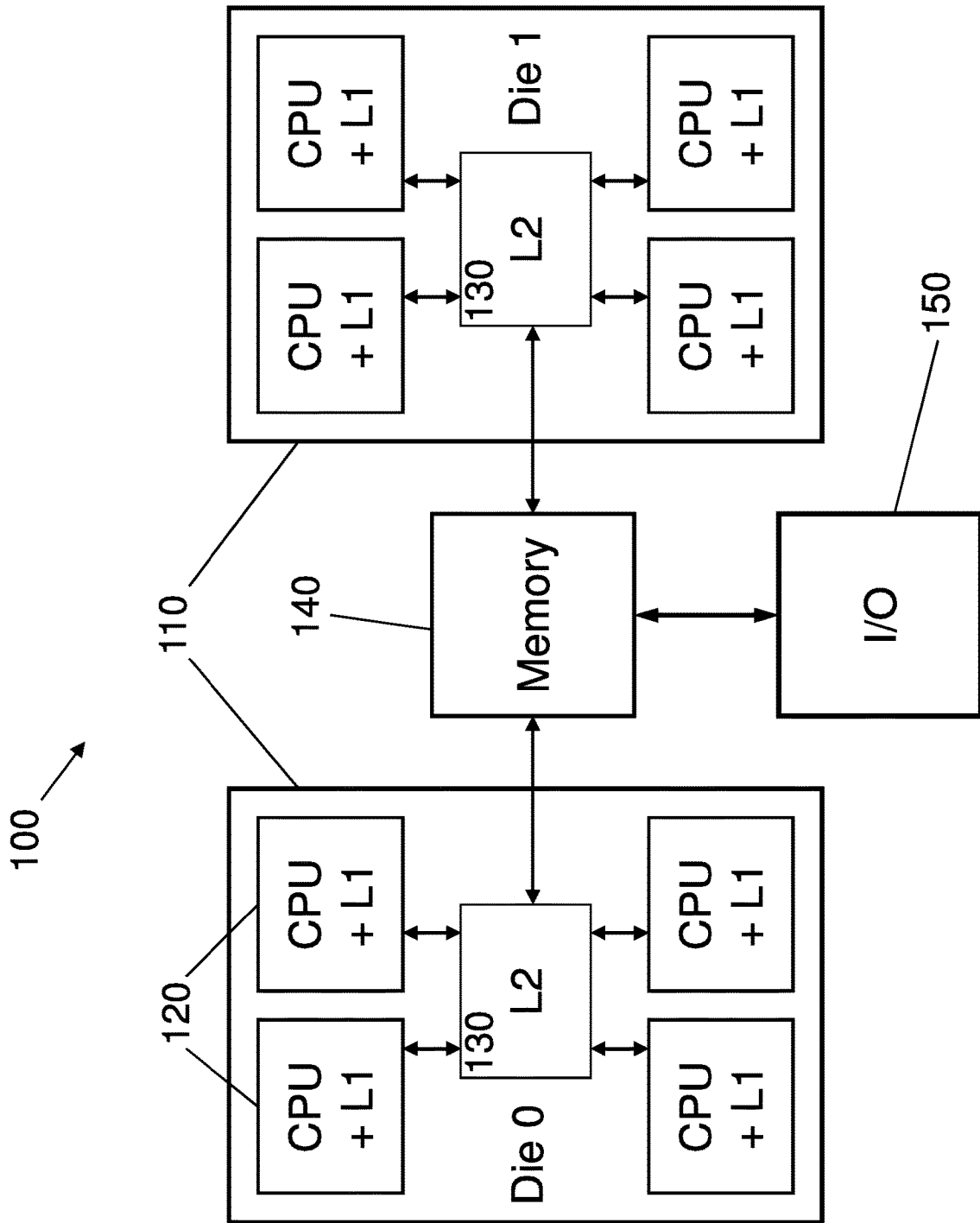
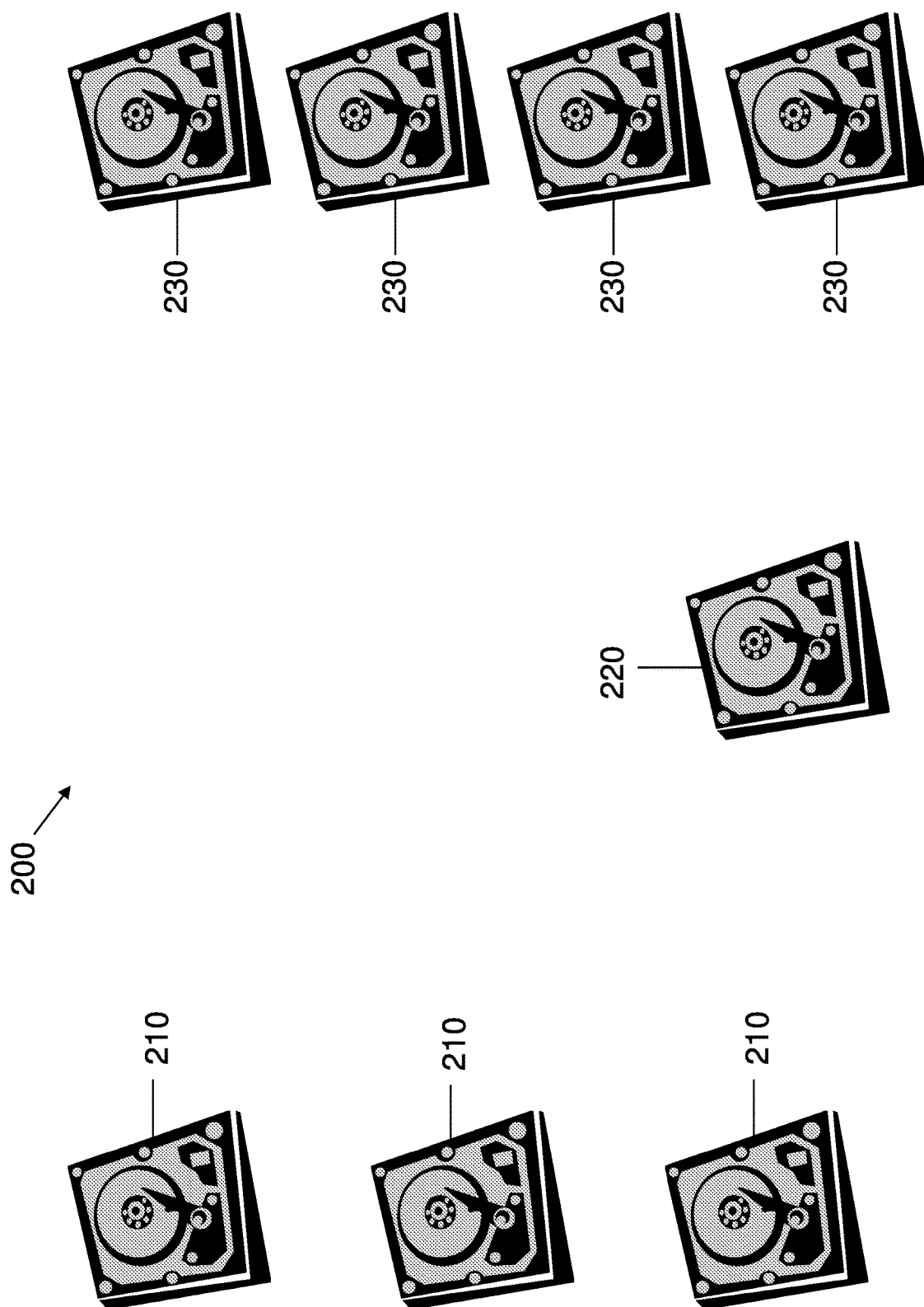


FIG. 9



US 10,666,296 B2

1

ACCELERATED ERASURE CODING SYSTEM AND METHOD

CROSS-REFERENCE TO RELATED APPLICATIONS

This application is a continuation of U.S. patent application Ser. No. 15/976,175 filed May 10, 2018, which is a continuation of U.S. patent application Ser. No. 15/201,196, filed on Jul. 1, 2016, now U.S. Pat. No. 10,003,358, issued on Jun. 19, 2018, which is a continuation of U.S. patent application Ser. No. 14/852,438, filed on Sep. 11, 2015, now U.S. Pat. No. 9,385,759, issued on Jul. 5, 2016, which is a continuation of U.S. patent application Ser. No. 14/223,740, filed on Mar. 24, 2014, now U.S. Pat. No. 9,160,374, issued on Oct. 13, 2015, which is a continuation of U.S. patent application Ser. No. 13/341,833, filed on Dec. 30, 2011, now U.S. Pat. No. 8,683,296, issued on Mar. 25, 2014, the entire contents of each of which are expressly incorporated herein by reference.

BACKGROUND

Field

Aspects of embodiments of the present invention are directed toward an accelerated erasure coding system and method.

Description of Related Art

An erasure code is a type of error-correcting code (ECC) useful for forward error-correction in applications like a redundant array of independent disks (RAID) or high-speed communication systems. In a typical erasure code, data (or original data) is organized in stripes, each of which is broken up into N equal-sized blocks, or data blocks, for some positive integer N . The data for each stripe is thus reconstructable by putting the N data blocks together. However, to handle situations where one or more of the original N data blocks gets lost, erasure codes also encode an additional M equal-sized blocks (called check blocks or check data) from the original N data blocks, for some positive integer M .

The N data blocks and the M check blocks are all the same size. Accordingly, there are a total of $N+M$ equal-sized blocks after encoding. The $N+M$ blocks may, for example, be transmitted to a receiver as $N+M$ separate packets, or written to $N+M$ corresponding disk drives. For ease of description, all $N+M$ blocks after encoding will be referred to as encoded blocks, though some (for example, N of them) may contain unencoded portions of the original data. That is, the encoded data refers to the original data together with the check data.

The M check blocks build redundancy into the system, in a very efficient manner, in that the original data (as well as any lost check data) can be reconstructed if any N of the $N+M$ encoded blocks are received by the receiver, or if any N of the $N+M$ disk drives are functioning correctly. Note that such an erasure code is also referred to as “optimal.” For ease of description, only optimal erasure codes will be discussed in this application. In such a code, up to M of the encoded blocks can be lost, (e.g., up to M of the disk drives can fail) so that if any N of the $N+M$ encoded blocks are received successfully by the receiver, the original data (as well as the check data) can be reconstructed. $N/(N+M)$ is thus the code rate of the erasure code encoding (i.e., how much space the original data takes up in the encoded data).

2

Erasure codes for select values of N and M can be implemented on RAID systems employing $N+M$ (disk) drives by spreading the original data among N “data” drives, and using the remaining M drives as “check” drives. Then, when any N of the $N+M$ drives are correctly functioning, the original data can be reconstructed, and the check data can be regenerated.

Erasure codes (or more specifically, erasure coding systems) are generally regarded as impractical for values of M larger than 1 (e.g., RAID5 systems, such as parity drive systems) or 2 (RAID6 systems), that is, for more than one or two check drives. For example, see H. Peter Anvin, “The mathematics of RAID-6,” the entire content of which is incorporated herein by reference, p. 7, “Thus, in 2-disk-degraded mode, performance will be very slow. However, it is expected that that will be a rare occurrence, and that performance will not matter significantly in that case.” See also Robert Maddock et al., “Surviving Two Disk Failures,” p. 6, “The main difficulty with this technique is that calculating the check codes, and reconstructing data after failures, is quite complex. It involves polynomials and thus multiplication, and requires special hardware, or at least a signal processor, to do it at sufficient speed.” In addition, see also James S. Plank, “All About Erasure Codes:—Reed-Solomon Coding—LDPC Coding,” slide 15 (describing computational complexity of Reed-Solomon decoding), “Bottom line: When n & m grow, it is brutally expensive.” Accordingly, there appears to be a general consensus among experts in the field that erasure coding systems are impractical for RAID systems for all but small values of M (that is, small numbers of check drives), such as 1 or 2.

Modern disk drives, on the other hand, are much less reliable than those envisioned when RAID was proposed. This is due to their capacity growing out of proportion to their reliability. Accordingly, systems with only a single check disk have, for the most part, been discontinued in favor of systems with two check disks.

In terms of reliability, a higher check disk count is clearly more desirable than a lower check disk count. If the count of error events on different drives is larger than the check disk count, data may be lost and that cannot be reconstructed from the correctly functioning drives. Error events extend well beyond the traditional measure of advertised mean time between failures (MTBF). A simple, real world example is a service event on a RAID system where the operator mistakenly replaces the wrong drive or, worse yet, replaces a good drive with a broken drive. In the absence of any generally accepted methodology to train, certify, and measure the effectiveness of service technicians, these types of events occur at an unknown rate, but certainly occur. The foolproof solution for protecting data in the face of multiple error events is to increase the check disk count.

SUMMARY

Aspects of embodiments of the present invention address these problems by providing a practical erasure coding system that, for byte-level RAID processing (where each byte is made up of 8 bits), performs well even for values of $N+M$ as large as 256 drives (for example, $N=127$ data drives and $M=129$ check drives). Further aspects provide for a single precomputed encoding matrix (or master encoding matrix) S of size $M_{max} \times N_{max}$, or $(N_{max} + M_{max}) \times N_{max}$ or $(M_{max} - 1) \times N_{max}$, elements (e.g., bytes), which can be used, for example, for any combination of $N \leq N_{max}$ data drives and $M \leq M_{max}$ check drives such that $N_{max} + M_{max} \leq 256$ (e.g., $N_{max}=127$ and $M_{max}=129$, or $N_{max}=63$ and $M_{max}=193$). This

US 10,666,296 B2

3

is an improvement over prior art solutions that rebuild such matrices from scratch every time N or M changes (such as adding another check drive). Still higher values of N and M are possible with larger processing increments, such as 2 bytes, which affords up to $N+M=65,536$ drives (such as $N=32,767$ data drives and $M=32,769$ check drives).

Higher check disk count can offer increased reliability and decreased cost. The higher reliability comes from factors such as the ability to withstand more drive failures. The decreased cost arises from factors such as the ability to create larger groups of data drives. For example, systems with two checks disks are typically limited to group sizes of 10 or fewer drives for reliability reasons. With a higher check disk count, larger groups are available, which can lead to fewer overall components for the same unit of storage and hence, lower cost.

Additional aspects of embodiments of the present invention further address these problems by providing a standard parity drive as part of the encoding matrix. For instance, aspects provide for a parity drive for configurations with up to 127 data drives and up to 128 (non-parity) check drives, for a total of up to 256 total drives including the parity drive. Further aspects provide for different breakdowns, such as up to 63 data drives, a parity drive, and up to 192 (non-parity) check drives. Providing a parity drive offers performance comparable to RAID5 in comparable circumstances (such as single data drive failures) while also being able to tolerate significantly larger numbers of data drive failures by including additional (non-parity) check drives.

Further aspects are directed to a system and method for implementing a fast solution matrix algorithm for Reed-Solomon codes. While known solution matrix algorithms compute an $N \times N$ solution matrix (see, for example, J. S. Plank, "A tutorial on Reed-Solomon coding for fault-tolerance in RAID-like systems," *Software—Practice & Experience*, 27(9):995-1012, September 1997, and J. S. Plank and Y. Ding, "Note: Correction to the 1997 tutorial on Reed-Solomon coding," Technical Report CS-03-504, University of Tennessee, April 2003), requiring $O(N^3)$ operations, regardless of the number of failed data drives, aspects of embodiments of the present invention compute only an $F \times F$ solution matrix, where F is the number of failed data drives. The overhead for computing this $F \times F$ solution matrix is approximately $F^3/3$ multiplication operations and the same number of addition operations. Not only is $F \leq N$, in almost any practical application, the number of failed data drives F is considerably smaller than the number of data drives N . Accordingly, the fast solution matrix algorithm is considerably faster than any known approach for practical values of F and N .

Still further aspects are directed toward fast implementations of the check data generation and the lost (original and check) data reconstruction. Some of these aspects are directed toward fetching the surviving (original and check) data a minimum number of times (that is, at most once) to carry out the data reconstruction. Some of these aspects are directed toward efficient implementations that can maximize or significantly leverage the available parallel processing power of multiple cores working concurrently on the check data generation and the lost data reconstruction. Existing implementations do not attempt to accelerate these aspects of the data generation and thus fail to achieve a comparable level of performance.

In an exemplary embodiment of the present invention, a system for accelerated error-correcting code (ECC) processing is provided. The system includes a processing core for executing computer instructions and accessing data from a

4

main memory; and a non-volatile storage medium (for example, a disk drive, or flash memory) for storing the computer instructions. The processing core, the storage medium, and the computer instructions are configured to implement an erasure coding system. The erasure coding system includes a data matrix for holding original data in the main memory, a check matrix for holding check data in the main memory, an encoding matrix for holding first factors in the main memory, and a thread for executing on the processing core. The first factors are for encoding the original data into the check data. The thread includes a parallel multiplier for concurrently multiplying multiple data entries of a matrix by a single factor; and a first sequencer for ordering operations through the data matrix and the encoding matrix using the parallel multiplier to generate the check data.

The first sequencer may be configured to access each entry of the data matrix from the main memory at most once while generating the check data.

The processing core may include a plurality of processing cores. The thread may include a plurality of threads. The erasure coding system may further include a scheduler for generating the check data by dividing the data matrix into a plurality of data matrices, dividing the check matrix into a plurality of check matrices, assigning corresponding ones of the data matrices and the check matrices to the threads, and assigning the threads to the processing cores to concurrently generate portions of the check data corresponding to the check matrices from respective ones of the data matrices.

The data matrix may include a first number of rows. The check matrix may include a second number of rows. The encoding matrix may include the second number of rows and the first number of columns.

The data matrix may be configured to add rows to the first number of rows or the check matrix may be configured to add rows to the second number of rows while the first factors remain unchanged.

Each of entries of one of the rows of the encoding matrix may include a multiplicative identity factor (such as 1).

The data matrix may be configured to be divided by rows into a surviving data matrix for holding surviving original data of the original data, and a lost data matrix corresponding to lost original data of the original data and including a third number of rows. The erasure coding system may further include a solution matrix for holding second factors in the main memory. The second factors are for decoding the check data into the lost original data using the surviving original data and the first factors.

The solution matrix may include the third number of rows and the third number of columns.

The solution matrix may further include an inverted said third number by said third number sub-matrix of the encoding matrix.

The erasure coding system may further include a first list of rows of the data matrix corresponding to the surviving data matrix, and a second list of rows of the data matrix corresponding to the lost data matrix.

The data matrix may be configured to be divided into a surviving data matrix for holding surviving original data of the original data, and a lost data matrix corresponding to lost original data of the original data. The erasure coding system may further include a solution matrix for holding second factors in the main memory. The second factors are for decoding the check data into the lost original data using the surviving original data and the first factors. The thread may further include a second sequencer for ordering operations through the surviving data matrix, the encoding matrix, the

US 10,666,296 B2

5

check matrix, and the solution matrix using the parallel multiplier to reconstruct the lost original data.

The second sequencer may be further configured to access each entry of the surviving data matrix from the main memory at most once while reconstructing the lost original data.

The processing core may include a plurality of processing cores. The thread may include a plurality of threads. The erasure coding system may further include: a scheduler for generating the check data and reconstructing the lost original data by dividing the data matrix into a plurality of data matrices; dividing the surviving data matrix into a plurality of surviving data matrices; dividing the lost data matrix into a plurality of lost data matrices; dividing the check matrix into a plurality of check matrices; assigning corresponding ones of the data matrices, the surviving data matrices, the lost data matrices, and the check matrices to the threads; and assigning the threads to the processing cores to concurrently generate portions of the check data corresponding to the check matrices from respective ones of the data matrices and to concurrently reconstruct portions of the lost original data corresponding to the lost data matrices from respective ones of the surviving data matrices and the check matrices.

The check matrix may be configured to be divided into a surviving check matrix for holding surviving check data of the check data, and a lost check matrix corresponding to lost check data of the check data. The second sequencer may be configured to order operations through the surviving data matrix, the reconstructed lost original data, and the encoding matrix using the parallel multiplier to regenerate the lost check data.

The second sequencer may be further configured to reconstruct the lost original data concurrently with regenerating the lost check data.

The second sequencer may be further configured to access each entry of the surviving data matrix from the main memory at most once while reconstructing the lost original data and regenerating the lost check data.

The second sequencer may be further configured to regenerate the lost check data without accessing the reconstructed lost original data from the main memory.

The processing core may include a plurality of processing cores. The thread may include a plurality of threads. The erasure coding system may further include a scheduler for generating the check data, reconstructing the lost original data, and regenerating the lost check data by: dividing the data matrix into a plurality of data matrices; dividing the surviving data matrix into a plurality of surviving data matrices; dividing the lost data matrix into a plurality of lost data matrices; dividing the check matrix into a plurality of check matrices; dividing the surviving check matrix into a plurality of surviving check matrices; dividing the lost check matrix into a plurality of lost check matrices; assigning corresponding ones of the data matrices, the surviving data matrices, the lost data matrices, the check matrices, the surviving check matrices, and the lost check matrices to the threads; and assigning the threads to the processing cores to concurrently generate portions of the check data corresponding to the check matrices from respective ones of the data matrices, to concurrently reconstruct portions of the lost original data corresponding to the lost data matrices from respective ones of the surviving data matrices and the surviving check matrices, and to concurrently regenerate portions of the lost check data corresponding to the lost check matrices from respective ones of the surviving data matrices and respective portions of the reconstructed lost original data.

6

The processing core may include 16 data registers. Each of the data registers may include 16 bytes. The parallel multiplier may be configured to process the data in units of at least 64 bytes spread over at least four of the data registers at a time.

Consecutive instructions to process each of the units of the data may access separate ones of the data registers to permit concurrent execution of the consecutive instructions by the processing core.

The parallel multiplier may include two lookup tables for doing concurrent multiplication of 4-bit quantities across 16 byte-sized entries using the PSHUFB (Packed Shuffle Bytes) instruction.

The parallel multiplier may be further configured to receive an input operand in four of the data registers, and return with the input operand intact in the four of the data registers.

According to another exemplary embodiment of the present invention, a method of accelerated error-correcting code (ECC) processing on a computing system is provided. The computing system includes a non-volatile storage medium (such as a disk drive or flash memory), a processing core for accessing instructions and data from a main memory, and a computer program including a plurality of computer instructions for implementing an erasure coding system. The method includes: storing the computer program on the storage medium; executing the computer instructions on the processing core; arranging original data as a data matrix in the main memory; arranging first factors as an encoding matrix in the main memory, the first factors being for encoding the original data into check data, the check data being arranged as a check matrix in the main memory; and generating the check data using a parallel multiplier for concurrently multiplying multiple data entries of a matrix by a single factor. The generating of the check data includes ordering operations through the data matrix and the encoding matrix using the parallel multiplier.

The generating of the check data may include accessing each entry of the data matrix from the main memory at most once.

The processing core may include a plurality of processing cores. The executing of the computer instructions may include executing the computer instructions on the processing cores. The method may further include scheduling the generating of the check data by: dividing the data matrix into a plurality of data matrices; dividing the check matrix into a plurality of check matrices; and assigning corresponding ones of the data matrices and the check matrices to the processing cores to concurrently generate portions of the check data corresponding to the check matrices from respective ones of the data matrices.

The method may further include: dividing the data matrix into a surviving data matrix for holding surviving original data of the original data, and a lost data matrix corresponding to lost original data of the original data; arranging second factors as a solution matrix in the main memory, the second factors being for decoding the check data into the lost original data using the surviving original data and the first factors; and reconstructing the lost original data by ordering operations through the surviving data matrix, the encoding matrix, the check matrix, and the solution matrix using the parallel multiplier.

The reconstructing of the lost original data may include accessing each entry of the surviving data matrix from the main memory at most once.

The processing core may include a plurality of processing cores. The executing of the computer instructions may

US 10,666,296 B2

7

include executing the computer instructions on the processing cores. The method may further include scheduling the generating of the check data and the reconstructing of the lost original data by: dividing the data matrix into a plurality of data matrices; dividing the surviving data matrix into a plurality of surviving data matrices; dividing the lost data matrix into a plurality of lost data matrices; dividing the check matrix into a plurality of check matrices; and assigning corresponding ones of the data matrices, the surviving data matrices, the lost data matrices, and the check matrices to the processing cores to concurrently generate portions of the check data corresponding to the check matrices from respective ones of the data matrices and to concurrently reconstruct portions of the lost original data corresponding to the lost data matrices from respective ones of the surviving data matrices and the check matrices.

The method may further include: dividing the check matrix into a surviving check matrix for holding surviving check data of the check data, and a lost check matrix corresponding to lost check data of the check data; and regenerating the lost check data by ordering operations through the surviving data matrix, the reconstructed lost original data, and the encoding matrix using the parallel multiplier.

The reconstructing of the lost original data may take place concurrently with the regenerating of the lost check data.

The reconstructing of the lost original data and the regenerating of the lost check data may include accessing each entry of the surviving data matrix from the main memory at most once.

The regenerating of the lost check data may take place without accessing the reconstructed lost original data from the main memory.

The processing core may include a plurality of processing cores. The executing of the computer instructions may include executing the computer instructions on the processing cores. The method may further include scheduling the generating of the check data, the reconstructing of the lost original data, and the regenerating of the lost check data by: dividing the data matrix into a plurality of data matrices; dividing the surviving data matrix into a plurality of surviving data matrices; dividing the lost data matrix into a plurality of lost data matrices; dividing the check matrix into a plurality of check matrices; dividing the surviving check matrix into a plurality of surviving check matrices; dividing the lost check matrix into a plurality of lost check matrices; and assigning corresponding ones of the data matrices, the surviving data matrices, the lost data matrices, the check matrices, the surviving check matrices, and the lost check matrices to the processing cores to concurrently generate portions of the check data corresponding to the check matrices from respective ones of the data matrices, to concurrently reconstruct portions of the lost original data corresponding to the lost data matrices from respective ones of the surviving data matrices and the surviving check matrices, and to concurrently regenerate portions of the lost check data corresponding to the lost check matrices from respective ones of the surviving data matrices and respective portions of the reconstructed lost original data.

According to yet another exemplary embodiment of the present invention, a non-transitory computer-readable storage medium (such as a disk drive, a compact disk (CD), a digital video disk (DVD), flash memory, a universal serial bus (USB) drive, etc.) containing a computer program including a plurality of computer instructions for performing accelerated error-correcting code (ECC) processing on a computing system is provided. The computing system

8

includes a processing core for accessing instructions and data from a main memory. The computer instructions are configured to implement an erasure coding system when executed on the computing system by performing the steps of: arranging original data as a data matrix in the main memory; arranging first factors as an encoding matrix in the main memory, the first factors being for encoding the original data into check data, the check data being arranged as a check matrix in the main memory; and generating the check data using a parallel multiplier for concurrently multiplying multiple data entries of a matrix by a single factor. The generating of the check data includes ordering operations through the data matrix and the encoding matrix using the parallel multiplier.

The generating of the check data may include accessing each entry of the data matrix from the main memory at most once.

The processing core may include a plurality of processing cores. The computer instructions may be further configured to perform the step of scheduling the generating of the check data by: dividing the data matrix into a plurality of data matrices; dividing the check matrix into a plurality of check matrices; and assigning corresponding ones of the data matrices and the check matrices to the processing cores to concurrently generate portions of the check data corresponding to the check matrices from respective ones of the data matrices.

The computer instructions may be further configured to perform the steps of: dividing the data matrix into a surviving data matrix for holding surviving original data of the original data, and a lost data matrix corresponding to lost original data of the original data; arranging second factors as a solution matrix in the main memory, the second factors being for decoding the check data into the lost original data using the surviving original data and the first factors; and reconstructing the lost original data by ordering operations through the surviving data matrix, the encoding matrix, the check matrix, and the solution matrix using the parallel multiplier.

The computer instructions may be further configured to perform the steps of: dividing the check matrix into a surviving check matrix for holding surviving check data of the check data, and a lost check matrix corresponding to lost check data of the check data; and regenerating the lost check data by ordering operations through the surviving data matrix, the reconstructed lost original data, and the encoding matrix using the parallel multiplier.

The reconstructing of the lost original data and the regenerating of the lost check data may include accessing each entry of the surviving data matrix from the main memory at most once.

The processing core may include a plurality of processing cores. The computer instructions may be further configured to perform the step of scheduling the generating of the check data, the reconstructing of the lost original data, and the regenerating of the lost check data by: dividing the data matrix into a plurality of data matrices; dividing the surviving data matrix into a plurality of surviving data matrices; dividing the lost data matrix into a plurality of lost data matrices; dividing the check matrix into a plurality of check matrices; dividing the surviving check matrix into a plurality of surviving check matrices; dividing the lost check matrix into a plurality of lost check matrices; and assigning corresponding ones of the data matrices, the surviving data matrices, the lost data matrices, the check matrices, the surviving check matrices, and the lost check matrices to the processing cores to concurrently generate portions of the

check data corresponding to the check matrices from respective ones of the data matrices, to concurrently reconstruct portions of the lost original data corresponding to the lost data matrices from respective ones of the surviving data matrices and the surviving check matrices, and to concurrently regenerate portions of the lost check data corresponding to the lost check matrices from respective ones of the surviving data matrices and respective portions of the reconstructed lost original data.

By providing practical and efficient systems and methods for erasure coding systems (which for byte-level processing can support up to $N+M=256$ drives, such as $N=127$ data drives and $M=129$ check drives, including a parity drive), applications such as RAID systems that can tolerate far more failing drives than was thought to be possible or practical can be implemented with accelerated performance significantly better than any prior art solution.

BRIEF DESCRIPTION OF THE DRAWINGS

The accompanying drawings, together with the specification, illustrate exemplary embodiments of the present invention and, together with the description, serve to explain aspects and principles of the present invention.

FIG. 1 shows an exemplary stripe of original and check data according to an embodiment of the present invention.

FIG. 2 shows an exemplary method for reconstructing lost data after a failure of one or more drives according to an embodiment of the present invention.

FIG. 3 shows an exemplary method for performing a parallel lookup Galois field multiplication according to an embodiment of the present invention.

FIG. 4 shows an exemplary method for sequencing the parallel lookup multiplier to perform the check data generation according to an embodiment of the present invention.

FIGS. 5-7 show an exemplary method for sequencing the parallel lookup multiplier to perform the lost data reconstruction according to an embodiment of the present invention.

FIG. 8 illustrates a multi-core architecture system according to an embodiment of the present invention.

FIG. 9 shows an exemplary disk drive configuration according to an embodiment of the present invention.

DETAILED DESCRIPTION

Hereinafter, exemplary embodiments of the invention will be described in more detail with reference to the accompanying drawings. In the drawings, like reference numerals refer to like elements throughout.

While optimal erasure codes have many applications, for ease of description, they will be described in this application with respect to RAID applications, i.e., erasure coding systems for the storage and retrieval of digital data distributed across numerous storage devices (or drives), though the present application is not limited thereto. For further ease of description, the storage devices will be assumed to be disk drives, though the invention is not limited thereto. In RAID systems, the data (or original data) is broken up into stripes, each of which includes N uniformly sized blocks (data blocks), and the N blocks are written across N separate drives (the data drives), one block per data drive.

In addition, for ease of description, blocks will be assumed to be composed of L elements, each element having a fixed size, say 8 bits or one byte. An element, such as a byte, forms the fundamental unit of operation for the RAID processing, but the invention is just as applicable to other

size elements, such as 16 bits (2 bytes). For simplification, unless otherwise indicated, elements will be assumed to be one byte in size throughout the description that follows, and the term “element(s)” and “byte(s)” will be used synonymously.

Conceptually, different stripes can distribute their data blocks across different combinations of drives, or have different block sizes or numbers of blocks, etc., but for simplification and ease of description and implementation, the described embodiments in the present application assume a consistent block size (L bytes) and distribution of blocks among the data drives between stripes. Further, all variables, such as the number of data drives N , will be assumed to be positive integers unless otherwise specified. In addition, since the $N=1$ case reduces to simple data mirroring (that is, copying the same data drive multiple times), it will also be assumed for simplicity that $N \geq 2$ throughout.

The N data blocks from each stripe are combined using arithmetic operations (to be described in more detail below) in M different ways to produce M blocks of check data (check blocks), and the M check blocks written across M drives (the check drives) separate from the N data drives, one block per check drive. These combinations can take place, for example, when new (or changed) data is written to (or back to) disk. Accordingly, each of the $N+M$ drives (data drives and check drives) stores a similar amount of data, namely one block for each stripe. As the processing of multiple stripes is conceptually similar to the processing of one stripe (only processing multiple blocks per drive instead of one), it will be further assumed for simplification that the data being stored or retrieved is only one stripe in size unless otherwise indicated. It will also be assumed that the block size L is sufficiently large that the data can be consistently divided across each block to produce subsets of the data that include respective portions of the blocks (for efficient concurrent processing by different processing units).

FIG. 1 shows an exemplary stripe 10 of original and check data according to an embodiment of the present invention.

Referring to FIG. 1, the stripe 10 can be thought of not only as the original N data blocks 20 that make up the original data, but also the corresponding M check blocks 30 generated from the original data (that is, the stripe 10 represents encoded data). Each of the N data blocks 20 is composed of L bytes 25 (labeled byte 1, byte 2, . . . , byte L), and each of the M check blocks 30 is composed of L bytes 35 (labeled similarly). In addition, check drive 1, byte 1, is a linear combination of data drive 1, byte 1; data drive 2, byte 1; . . . ; data drive N , byte 1. Likewise, check drive 1, byte 2, is generated from the same linear combination formula as check drive 1, byte 1, only using data drive 1, byte 2; data drive 2, byte 2; . . . ; data drive N , byte 2. In contrast, check drive 2, byte 1, uses a different linear combination formula than check drive 1, byte 1, but applies it to the same data, namely data drive 1, byte 1; data drive 2, byte 1; . . . ; data drive N , byte 1. In this fashion, each of the other check bytes 35 is a linear combination of the respective bytes of each of the N data drives 20 and using the corresponding linear combination formula for the particular check drive 30.

The stripe 10 in FIG. 1 can also be represented as a matrix C of encoded data. C has two sub-matrices, namely original data D on top and check data J on bottom. That is,

11

$$C = \begin{bmatrix} D \\ J \end{bmatrix} = \begin{bmatrix} D_{11} & D_{12} & \dots & D_{1L} \\ D_{21} & D_{22} & \dots & D_{2L} \\ \vdots & \vdots & \ddots & \vdots \\ D_{N1} & D_{N2} & \dots & D_{NL} \\ J_{11} & J_{12} & \dots & J_{1L} \\ J_{21} & J_{22} & \dots & J_{2L} \\ \vdots & \vdots & \ddots & \vdots \\ J_{M1} & J_{M2} & \dots & J_{ML} \end{bmatrix},$$

where D_{ij} =byte j from data drive i and J_{ij} =byte j from check drive i. Thus, the rows of encoded data C represent blocks, while the columns represent corresponding bytes of each of the drives.

Further, in case of a disk drive failure of one or more disks, the arithmetic operations are designed in such a fashion that for any stripe, the original data (and by extension, the check data) can be reconstructed from any combination of N data and check blocks from the corresponding N+M data and check blocks that comprise the stripe. Thus, RAID provides both parallel processing (reading and writing the data in stripes across multiple drives concurrently) and fault tolerance (regeneration of the original data even if as many as M of the drives fail), at the computational cost of generating the check data any time new data is written to disk, or changed data is written back to disk, as well as the computational cost of reconstructing any lost original data and regenerating any lost check data after a disk failure.

For example, for M=1 check drive, a single parity drive can function as the check drive (i.e., a RAID4 system). Here, the arithmetic operation is bitwise exclusive OR of each of the N corresponding data bytes in each data block of the stripe. In addition, as mentioned earlier, the assignment of parity blocks from different stripes to the same drive (i.e., RAID4) or different drives (i.e., RAID5) is arbitrary, but it does simplify the description and implementation to use a consistent assignment between stripes, so that will be assumed throughout. Since M=1 reduces to the case of a single parity drive, it will further be assumed for simplicity that $M \geq 2$ throughout.

For such larger values of M, Galois field arithmetic is used to manipulate the data, as described in more detail later. Galois field arithmetic, for Galois fields of powers-of-2 (such as 2^p) numbers of elements, includes two fundamental operations: (1) addition (which is just bitwise exclusive OR, as with the parity drive-only operations for M=1), and (2) multiplication. While Galois field (GF) addition is trivial on standard processors, GF multiplication is not. Accordingly, a significant component of RAID performance for $M \geq 2$ is speeding up the performance of GF multiplication, as will be discussed later. For purposes of description, GF addition will be represented by the symbol+throughout while GF multiplication will be represented by the symbolxthroughout.

Briefly, in exemplary embodiments of the present invention, each of the M check drives holds linear combinations (over GF arithmetic) of the N data drives of original data, one linear combination (i.e., a GF sum of N terms, where each term represents a byte of original data times a corresponding factor (using GF multiplication) for the respective data drive) for each check drive, as applied to respective bytes in each block. One such linear combination can be a simple parity, i.e., entirely GF addition (all factors equal 1), such as a GF sum of the first byte in each block of original data as described above.

12

The remaining M-1 linear combinations include more involved calculations that include the nontrivial GF multiplication operations (e.g., performing a GF multiplication of the first byte in each block by a corresponding factor for the respective data drive, and then performing a GF sum of all these products). These linear combinations can be represented by an (N+M)xN matrix (encoding matrix or information dispersal matrix (IDM)) E of the different factors, one factor for each combination of (data or check) drive and data drive, with one row for each of the N+M data and check drives and one column for each of the N data drives. The IDM E can also be represented as

$$\begin{bmatrix} I_N \\ H \end{bmatrix},$$

where I_N represents the NxN identity matrix (i.e., the original (unencoded) data) and H represents the MxN matrix of factors for the check drives (where each of the M rows corresponds to one of the M check drives and each of the N columns corresponds to one of the N data drives).

Thus,

$$E = \begin{bmatrix} I_N \\ H \end{bmatrix} = \begin{bmatrix} 1 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 1 \\ H_{11} & H_{12} & \dots & H_{1N} \\ H_{21} & H_{22} & \dots & H_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ H_{M1} & H_{M2} & \dots & H_{MN} \end{bmatrix},$$

where H_{ij} =factor for check drive i and data drive j. Thus, the rows of encoded data C represent blocks, while the columns represent corresponding bytes of each of the drives. In addition, check factors H, original data D, and check data J are related by the formula $J=H \times D$ (that is, matrix multiplication), or

$$\begin{bmatrix} J_{11} & J_{12} & \dots & J_{1L} \\ J_{21} & J_{22} & \dots & J_{2L} \\ \vdots & \vdots & \ddots & \vdots \\ J_{M1} & J_{M2} & \dots & J_{ML} \end{bmatrix} = \begin{bmatrix} H_{11} & H_{12} & \dots & H_{1N} \\ H_{21} & H_{22} & \dots & H_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ H_{M1} & H_{M2} & \dots & H_{MN} \end{bmatrix} \times \begin{bmatrix} D_{11} & D_{12} & \dots & D_{1L} \\ D_{21} & D_{22} & \dots & D_{2L} \\ \vdots & \vdots & \ddots & \vdots \\ D_{N1} & D_{N2} & \dots & D_{NL} \end{bmatrix},$$

where $J_{11}=(H_{11} \times D_{11})+(H_{12} \times D_{21})+ \dots +(H_{1N} \times D_{N1})$, $J_{12}=(H_{11} \times D_{12})+(H_{12} \times D_{22})+ \dots +(H_{1N} \times D_{N2})$, $J_{21}=(H_{21} \times D_{11})+(H_{22} \times D_{21})+ \dots +(H_{2N} \times D_{N1})$, and in general, $J_{ij}=(H_{i1} \times D_{1j})+(H_{i2} \times D_{2j})+ \dots +(H_{iN} \times D_{Nj})$ for $1 \leq i \leq M$ and $1 \leq j \leq L$.

Such an encoding matrix E is also referred to as an information dispersal matrix (IDM). It should be noted that matrices such as check drive encoding matrix H and identity matrix I_N also represent encoding matrices, in that they represent matrices of factors to produce linear combinations over GF arithmetic of the original data. In practice, the identity matrix I_N is trivial and may not need to be con-

structured as part of the IDM E. Only the encoding matrix E, however, will be referred to as the IDM. Methods of building an encoding matrix such as IDM E or check drive encoding matrix H are discussed below. In further embodiments of the present invention (as discussed further in Appendix A), such (N+M)×N (or M×N) matrices can be trivially constructed (or simply indexed) from a master encoding matrix S, which is composed of (N_{max}+M_{max})×N_{max} (or M_{max}×N_{max}) bytes or elements, where N_{max}+M_{max}=256 (or some other power of two) and N≤N_{max} and M≤M_{max}. For example, one such master encoding matrix S can include a 127×127 element identity matrix on top (for up to N_{max}=127 data drives), a row of 1's (for a parity drive), and a 128×127 element encoding matrix on bottom (for up to M_{max}=129 check drives, including the parity drive), for a total of N_{max}+M_{max}=256 drives.

The original data, in turn, can be represented by an N×L matrix D of bytes, each of the N rows representing the L bytes of a block of the corresponding one of the N data drives. If C represents the corresponding (N+M)×L matrix of encoded bytes (where each of the N+M rows corresponds to one of the N+M data and check drives), then C can be represented as

$$E \times D = \begin{bmatrix} I_N \\ H \end{bmatrix} \times D = \begin{bmatrix} I_N \times D \\ H \times D \end{bmatrix} = \begin{bmatrix} D \\ J \end{bmatrix},$$

where J=H×D is an M×L matrix of check data, with each of the M rows representing the L check bytes of the corresponding one of the M check drives. It should be noted that in the relationships such as C=E×D or J=H×D, x represents matrix multiplication over the Galois field (i.e., GF multiplication and GF addition being used to generate each of the entries in, for example, C or

In exemplary embodiments of the present invention, the first row of the check drive encoding matrix H (or the (N+1)th row of the IDM E) can be all 1's, representing the parity drive. For linear combinations involving this row, the GF multiplication can be bypassed and replaced with a GF sum of the corresponding bytes since the products are all trivial products involving the identity element 1. Accordingly, in parity drive implementations, the check drive encoding matrix H can also be thought of as an (M-1)×N matrix of non-trivial factors (that is, factors intended to be used in GF multiplication and not just GF addition).

Much of the RAID processing involves generating the check data when new or changed data is written to (or back to) disk. The other significant event for RAID processing is when one or more of the drives fail (data or check drives), or for whatever reason become unavailable. Assume that in such a failure scenario, F data drives fail and G check drives fail, where F and G are nonnegative integers. If F=0, then only check drives failed and all of the original data D survived. In this case, the lost check data can be regenerated from the original data D.

Accordingly, assume at least one data drive fails, that is, F≥1, and let K=N-F represent the number of data drives that survive. K is also a nonnegative integer. In addition, let X represent the surviving original data and Y represent the lost original data. That is, X is a K×L matrix composed of the K rows of the original data matrix D corresponding to the K surviving data drives, while Y is an F×L matrix composed of the F rows of the original data matrix D corresponding to the F failed data drives.

$$\begin{bmatrix} X \\ Y \end{bmatrix}$$

thus represents a permuted original data matrix D' (that is, the original data matrix D, only with the surviving original data X on top and the lost original data Y on bottom. It should be noted that once the lost original data Y is reconstructed, it can be combined with the surviving original data X to restore the original data D, from which the check data for any of the failed check drives can be regenerated.

It should also be noted that M-G check drives survive. In order to reconstruct the lost original data Y, enough (that is, at least N) total drives must survive. Given that K=N-F data drives survive, and that M-G check drives survive, it follows that (N-F)+(M-G)≥N must be true to reconstruct the lost original data Y. This is equivalent to F+G≤M (i.e., no more than F+G drives fail), or F≤M-G (that is, the number of failed data drives does not exceed the number of surviving check drives). It will therefore be assumed for simplicity that F≤M-G.

In the routines that follow, performance can be enhanced by prebuilding lists of the failed and surviving data and check drives (that is, four separate lists). This allows processing of the different sets of surviving and failed drives to be done more efficiently than existing solutions, which use, for example, bit vectors that have to be examined one bit at a time and often include large numbers of consecutive zeros (or ones) when ones (or zeros) are the bit values of interest.

FIG. 2 shows an exemplary method 300 for reconstructing lost data after a failure of one or more drives according to an embodiment of the present invention.

While the recovery process is described in more detail later, briefly it consists of two parts: (1) determining the solution matrix, and (2) reconstructing the lost data from the surviving data. Determining the solution matrix can be done in three steps with the following algorithm (Algorithm 1), with reference to FIG. 2:

1. (Step 310 in FIG. 2) Reducing the (M+N)×N IDM E to an N×N reduced encoding matrix T (also referred to as the transformed IDM) including the K surviving data drive rows and any F of the M-G surviving check drive rows (for instance, the first F surviving check drive rows, as these will include the parity drive if it survived; recall that F≤M-G was assumed). In addition, the columns of the reduced encoding matrix T are rearranged so that the K columns corresponding to the K surviving data drives are on the left side of the matrix and the F columns corresponding to the F failed drives are on the right side of the matrix. (Step 320) These F surviving check drives selected to rebuild the lost original data Y will henceforth be referred to as "the F surviving check drives," and their check data W will be referred to as "the surviving check data," even though M-G check drives survived. It should be noted that W is an F×L matrix composed of the F rows of the check data J corresponding to the F surviving check drives. Further, the surviving encoded data can be represented as a sub-matrix C' of the encoded data C. The surviving encoded data C' is an N×L matrix composed of the surviving original data X on top and the surviving check data W on bottom, that is,

$$C' = \begin{bmatrix} X \\ W \end{bmatrix}.$$

15

2. (Step 330) Splitting the reduced encoding matrix T into four sub-matrices (that are also encoding matrices): (i) a K×K identity matrix I_K (corresponding to the K surviving data drives) in the upper left, (ii) a K×F matrix O of zeros in the upper right, (iii) an F×K encoding matrix A in the lower left corresponding to the F surviving check drive rows and the K surviving data drive columns, and (iv) an F×F encoding matrix B in the lower right corresponding to the F surviving check drive rows and the F failed data drive columns. Thus, the reduced encoding matrix T can be represented as

$$\begin{bmatrix} I_K & O \\ A & B \end{bmatrix}$$

3. (Step 340) Calculating the inverse B^{-1} of the F×F encoding matrix B. As is shown in more detail in Appendix A, $C'=TxD'$, or

$$\begin{bmatrix} X \\ W \end{bmatrix} = \begin{bmatrix} I_K & O \\ A & B \end{bmatrix} \times \begin{bmatrix} X \\ Y \end{bmatrix},$$

which is mathematically equivalent to $W=A \times X+B \times Y$. B^{-1} is the solution matrix, and is itself an F×F encoding matrix. Calculating the solution matrix B^{-1} thus allows the lost original data Y to be reconstructed from the encoding matrices A and B along with the surviving original data X and the surviving check data W.

The F×K encoding matrix A represents the original encoding matrix E, only limited to the K surviving data drives and the F surviving check drives. That is, each of the F rows of A represents a different one of the F surviving check drives, while each of the K columns of A represents a different one of the K surviving data drives. Thus, A provides the encoding factors needed to encode the original data for the surviving check drives, but only applied to the surviving data drives (that is, the surviving partial check data). Since the surviving original data X is available, A can be used to generate this surviving partial check data.

In similar fashion, the F×F encoding matrix B represents the original encoding matrix E, only limited to the F surviving check drives and the F failed data drives. That is, the F rows of B correspond to the same F rows of A, while each of the F columns of B represents a different one of the F failed data drives. Thus, B provides the encoding factors needed to encode the original data for the surviving check drives, but only applied to the failed data drives (that is, the lost partial check data). Since the lost original data Y is not available, B cannot be used to generate any of the lost partial check data. However, this lost partial check data can be determined from A and the surviving check data W. Since this lost partial check data represents the result of applying B to the lost original data Y, B^{-1} thus represents the necessary factors to reconstruct the lost original data Y from the lost partial check data.

It should be noted that steps 1 and 2 in Algorithm 1 above are logical, in that encoding matrices A and B (or the reduced encoding matrix T, for that matter) do not have to actually be constructed. Appropriate indexing of the IDM E (or the master encoding matrix S) can be used to obtain any of their entries. Step 3, however, is a matrix inversion over GF arithmetic and takes $O(F^3)$ operations, as discussed in more

16

detail later. Nonetheless, this is a significant improvement over existing solutions, which require $O(N^3)$ operations, since the number of failed data drives F is usually significantly less than the number of data drives N in any practical situation.

(Step 350 in FIG. 2) Once the encoding matrix A and the solution matrix B^{-1} are known, reconstructing the lost data from the surviving data (that is, the surviving original data X and the surviving check data W) can be accomplished in four steps using the following algorithm (Algorithm 2):

1. Use A and the surviving original data X (using matrix multiplication) to generate the surviving check data (i.e., $A \times X$), only limited to the K surviving data drives. Call this limited check data the surviving partial check data.
2. Subtract this surviving partial check data from the surviving check data W (using matrix subtraction, i.e., $W-A \times X$, which is just entry-by-entry GF subtraction, which is the same as GF addition for this Galois field). This generates the surviving check data, only this time limited to the F failed data drives. Call this limited check data the lost partial check data.
3. Use the solution matrix B^{-1} and the lost partial check data (using matrix multiplication, i.e., $B^{-1} \times (W-A \times X)$) to reconstruct the lost original data Y. Call this the recovered original data Y.
4. Use the corresponding rows of the IDM E (or master encoding matrix 5) for each of the G failed check drives along with the original data D, as reconstructed from the surviving and recovered original data X and Y, to regenerate the lost check data (using matrix multiplication).

As will be shown in more detail later, steps 1-3 together require $O(F)$ operations times the amount of original data D to reconstruct the lost original data Y for the F failed data drives (i.e., roughly 1 operation per failed data drive per byte of original data D), which is proportionally equivalent to the $O(M)$ operations times the amount of original data D needed to generate the check data J for the M check drives (i.e., roughly 1 operation per check drive per byte of original data D). In addition, this same equivalence extends to step 4, which takes $O(G)$ operations times the amount of original data D needed to regenerate the lost check data for the G failed check drives (i.e., roughly 1 operation per failed check drive per byte of original data D). In summary, the number of operations needed to reconstruct the lost data is $O(F+G)$ times the amount of original data D (i.e., roughly 1 operation per failed drive (data or check) per byte of original data D). Since $F+G \leq M$, this means that the computational complexity of Algorithm 2 (reconstructing the lost data from the surviving data) is no more than that of generating the check data J from the original data D.

As mentioned above, for exemplary purposes and ease of description, data is assumed to be organized in 8-bit bytes, each byte capable of taking on $2^8=256$ possible values. Such data can be manipulated in byte-size elements using GF arithmetic for a Galois field of size $2^8=256$ elements. It should also be noted that the same mathematical principles apply to any power-of-two 2^P number of elements, not just 256, as Galois fields can be constructed for any integral power of a prime number. Since Galois fields are finite, and since GF operations never overflow, all results are the same size as the inputs, for example, 8 bits.

In a Galois field of a power-of-two number of elements, addition and subtraction are the same operation, namely a bitwise exclusive OR (XOR) of the two operands. This is a very fast operation to perform on any current processor. It

can also be performed on multiple bytes concurrently. Since the addition and subtraction operations take place, for example, on a byte-level basis, they can be done in parallel by using, for instance, x86 architecture Streaming SIMD Extensions (SSE) instructions (SIMD stands for single instruction, multiple data, and refers to performing the same instruction on different pieces of data, possibly concurrently), such as PXOR (Packed (bitwise) Exclusive OR).

SSE instructions can process, for example, 16-byte registers (XMM registers), and are able to process such registers as though they contain 16 separate one-byte operands (or 8 separate two-byte operands, or four separate four-byte operands, etc.) Accordingly, SSE instructions can do byte-level processing 16 times faster than when compared to processing a byte at a time. Further, there are 16 XMM registers, so dedicating four such registers for operand storage allows the data to be processed in 64-byte increments, using the other 12 registers for temporary storage. That is, individual operations can be performed as four consecutive SSE operations on the four respective registers (64 bytes), which can often allow such instructions to be efficiently pipelined and/or concurrently executed by the processor. In addition, the SSE instructions allows the same processing to be performed on different such 64-byte increments of data in parallel using different cores. Thus, using four separate cores can potentially speed up this processing by an additional factor of 4 over using a single core.

For example, a parallel adder (Parallel Adder) can be built using the 16-byte XMM registers and four consecutive PXOR instructions. Such parallel processing (that is, 64 bytes at a time with only a few machine-level instructions) for GF arithmetic is a significant improvement over doing the addition one byte at a time. Since the data is organized in blocks of any fixed number of bytes, such as 4096 bytes (4 kilobytes, or 4 KB) or 32,768 bytes (32 KB), a block can be composed of numerous such 64-byte chunks (e.g., 64 separate 64-byte chunks in 4 KB, or 512 chunks in 32 KB).

Multiplication in a Galois field is not as straightforward. While much of it is bitwise shifts and exclusive OR's (i.e., "additions") that are very fast operations, the numbers "wrap" in peculiar ways when they are shifted outside of their normal bounds (because the field has only a finite set of elements), which can slow down the calculations. This "wrapping" in the GF multiplication can be addressed in many ways. For example, the multiplication can be implemented serially (Serial Multiplier) as a loop iterating over the bits of one operand while performing the shifts, adds, and wraps on the other operand. Such processing, however, takes several machine instructions per bit for 8 separate bits. In other words, this technique requires dozens of machine instructions per byte being multiplied. This is inefficient compared to, for example, the performance of the Parallel Adder described above.

For another approach (Serial Lookup Multiplier), multiplication tables (of all the possible products, or at least all the non-trivial products) can be pre-computed and built ahead of time. For example, a table of $256 \times 256 = 65,536$ bytes can hold all the possible products of the two different one-byte operands). However, such tables can force serialized access on what are only byte-level operations, and not take advantage of wide (concurrent) data paths available on modern processors, such as those used to implement the Parallel Adder above.

In still another approach (Parallel Multiplier), the GF multiplication can be done on multiple bytes at a time, since the same factor in the encoding matrix is multiplied with every element in a data block. Thus, the same factor can be

multiplied with 64 consecutive data block bytes at a time. This is similar to the Parallel Adder described above, only there are several more operations needed to perform the operation. While this can be implemented as a loop on each bit of the factor, as described above, only performing the shifts, adds, and wraps on 64 bytes at a time, it can be more efficient to process the 256 possible factors as a (C language) switch statement, with inline code for each of 256 different combinations of two primitive GF operations: Multiply-by-2 and Add. For example, GF multiplication by the factor 3 can be effected by first doing a Multiply-by-2 followed by an Add. Likewise, GF multiplication by 4 is just a Multiply-by-2 followed by a Multiply-by-2 while multiplication by 6 is a Multiply-by-2 followed by an Add and then by another Multiply-by-2.

While this Add is identical to the Parallel Adder described above (e.g., four consecutive PXOR instructions to process 64 separate bytes), Multiply-by-2 is not as straightforward. For example, Multiply-by-2 in GF arithmetic can be implemented across 64 bytes at a time in 4 XMM registers via 4 consecutive PXOR instructions, 4 consecutive PCMPGTB (Packed Compare for Greater Than) instructions, 4 consecutive PADDB (Packed Add) instructions, 4 consecutive PAND (Bitwise AND) instructions, and 4 consecutive PXOR instructions. Though this takes 20 machine instructions, the instructions are very fast and results in 64 consecutive bytes of data at a time being multiplied by 2.

For 64 bytes of data, assuming a random factor between 0 and 255, the total overhead for the Parallel Multiplier is about 6 calls to multiply-by-2 and about 3.5 calls to add, or about $6 \times 20 + 3.5 \times 4 = 134$ machine instructions, or a little over 2 machine instructions per byte of data. While this compares favorably with byte-level processing, it is still possible to improve on this by building a parallel multiplier with a table lookup (Parallel Lookup Multiplier) using the PSHUFB (Packed Shuffle Bytes) instruction and doing the GF multiplication in 4-bit nibbles (half bytes).

FIG. 3 shows an exemplary method 400 for performing a parallel lookup Galois field multiplication according to an embodiment of the present invention.

Referring to FIG. 3, in step 410, two lookup tables are built once: one lookup table for the low-order nibbles in each byte, and one lookup table for the high-order nibbles in each byte. Each lookup table contains 256 sets (one for each possible factor) of the 16 possible GF products of that factor and the 16 possible nibble values. Each lookup table is thus $256 \times 16 = 4096$ bytes, which is considerably smaller than the 65,536 bytes needed to store a complete one-byte multiplication table. In addition, PSHUFB does 16 separate table lookups at once, each for one nibble, so 8 PSHUFB instructions can be used to do all the table lookups for 64 bytes (128 nibbles).

Next, in step 420, the Parallel Lookup Multiplier is initialized for the next set of 64 bytes of operand data (such as original data or surviving original data). In order to save loading this data from memory on succeeding calls, the Parallel Lookup Multiplier dedicates four registers for this data, which are left intact upon exit of the Parallel Lookup Multiplier. This allows the same data to be called with different factors (such as processing the same data for another check drive).

Next in step 430, to process these 64 bytes of operand data, the Parallel Lookup Multiplier can be implemented with 2 MOVDQA (Move Double Quadword Aligned) instructions (from memory) to do the two table lookups and 4 MOVDQA instructions (register to register) to initialize registers (such as the output registers). These are followed in

US 10,666,296 B2

19

steps 440 and 450 by two nearly identical sets of 17 register-to-register instructions to carry out the multiplication 32 bytes at a time. Each such set starts (in step 440) with 5 more MOVDQA instructions for further initialization, followed by 2 PSRLW (Packed Shift Right Logical Word) instructions to realign the high-order nibbles for PSHUFB, and 4 PAND instructions to clear the high-order nibbles for PSHUFB. That is, two registers of byte operands are converted into four registers of nibble operands. Then, in step 450, 4 PSHUFB instructions are used to do the parallel table lookups, and 2 PXOR instructions to add the results of the multiplication on the two nibbles to the output registers.

Thus, the Parallel Lookup Multiplier uses 40 machine instructions to perform the parallel multiplication on 64 separate bytes, which is considerably better than the average 134 instructions for the Parallel Multiplier above, and only 10 times as many instructions as needed for the Parallel Adder. While some of the Parallel Lookup Multiplier's instructions are more complex than those of the Parallel Adder, much of this complexity can be concealed through the pipelined and/or concurrent execution of numerous such contiguous instructions (accessing different registers) on modern pipelined processors. For example, in exemplary implementations, the Parallel Lookup Multiplier has been timed at about 15 CPU clock cycles per 64 bytes processed per CPU core (about 0.36 clock cycles per instruction). In addition, the code footprint is practically nonexistent for the Parallel Lookup Multiplier (40 instructions) compared to that of the Parallel Multiplier (about 34,300 instructions), even when factoring the 8 KB needed for the two lookup tables in the Parallel Lookup Multiplier.

In addition, embodiments of the Parallel Lookup Multiplier can be passed 64 bytes of operand data (such as the next 64 bytes of surviving original data X to be processed) in four consecutive registers, whose contents can be preserved upon exiting the Parallel Lookup Multiplier (and all in the same 40 machine instructions) such that the Parallel Lookup Multiplier can be invoked again on the same 64 bytes of data without having to access main memory to reload the data. Through such a protocol, memory accesses can be minimized (or significantly reduced) for accessing the original data D during check data generation or the surviving original data X during lost data reconstruction.

Further embodiments of the present invention are directed towards sequencing this parallel multiplication (and other GF) operations. While the Parallel Lookup Multiplier processes a GF multiplication of 64 bytes of contiguous data times a specified factor, the calls to the Parallel Lookup Multiplier should be appropriately sequenced to provide efficient processing. One such sequencer (Sequencer 1), for example, can generate the check data J from the original data D, and is described further with respect to FIG. 4.

The parity drive does not need GF multiplication. The check data for the parity drive can be obtained, for example, by adding corresponding 64-byte chunks for each of the data drives to perform the parity operation. The Parallel Adder can do this using 4 instructions for every 64 bytes of data for each of the N data drives, or N/16 instructions per byte.

The M-1 non-parity check drives can invoke the Parallel Lookup Multiplier on each 64-byte chunk, using the appropriate factor for the particular combination of data drive and check drive. One consideration is how to handle the data access. Two possible ways are:

- 1) "column-by-column," i.e., 64 bytes for one data drive, followed by the next 64 bytes for that data drive, etc.,

20

and adding the products to the running total in memory (using the Parallel Adder) before moving onto the next row (data drive); and

- 2) "row-by-row," i.e., 64 bytes for one data drive, followed by the corresponding 64 bytes for the next data drive, etc., and keeping a running total using the Parallel Adder, then moving onto the next set of 64-byte chunks.

Column-by-column can be thought of as "constant factor, varying data," in that the (GF multiplication) factor usually remains the same between iterations while the (64-byte) data changes with each iteration. Conversely, row-by-row can be thought of as "constant data, varying factor," in that the data usually remains the same between iterations while the factor changes with each iteration.

Another consideration is how to handle the check drives. Two possible ways are:

- a) one at a time, i.e., generate all the check data for one check drive before moving onto the next check drive; and
- b) all at once, i.e., for each 64-byte chunk of original data, do all of the processing for each of the check drives before moving onto the next chunk of original data.

While each of these techniques performs the same basic operations (e.g., 40 instructions for every 64 bytes of data for each of the N data drives and M-1 non-parity check drives, or $5N(M-1)/8$ instructions per byte for the Parallel Lookup Multiplier), empirical results show that combination (2)(b), that is, row-by-row data access on all of the check drives between data accesses performs best with the Parallel Lookup Multiplier. One reason may be that such an approach appears to minimize the number of memory accesses (namely, one) to each chunk of the original data D to generate the check data J. This embodiment of Sequencer 1 is described in more detail with reference to FIG. 4.

FIG. 4 shows an exemplary method 500 for sequencing the Parallel Lookup Multiplier to perform the check data generation according to an embodiment of the present invention.

Referring to FIG. 4, in step 510, the Sequencer 1 is called. Sequencer 1 is called to process multiple 64-byte chunks of data for each of the blocks across a stripe of data. For instance, Sequencer 1 could be called to process 512 bytes from each block. If, for example, the block size L is 4096 bytes, then it would take eight such calls to Sequencer 1 to process the entire stripe. The other such seven calls to Sequencer 1 could be to different processing cores, for instance, to carry out the check data generation in parallel. The number of 64-byte chunks to process at a time could depend on factors such as cache dimensions, input/output data structure sizes, etc.

In step 520, the outer loop processes the next 64-byte chunk of data for each of the drives. In order to minimize the number of accesses of each data drive's 64-byte chunk of data from memory, the data is loaded only once and preserved across calls to the Parallel Lookup Multiplier. The first data drive is handled specially since the check data has to be initialized for each check drive. Using the first data drive to initialize the check data saves doing the initialization as a separate step followed by updating it with the first data drive's data. In addition to the first data drive, the first check drive is also handled specially since it is a parity drive, so its check data can be initialized to the first data drive's data directly without needing the Parallel Lookup Multiplier.

In step 530, the first middle loop is called, in which the remainder of the check drives (that is, the non-parity check drives) have their check data initialized by the first data

US 10,666,296 B2

21

drive's data. In this case, there is a corresponding factor (that varies with each check drive) that needs to be multiplied with each of the first data drive's data bytes. This is handled by calling the Parallel Lookup Multiplier for each non-parity check drive.

In step 540, the second middle loop is called, which processes the other data drives' corresponding 64-byte chunks of data. As with the first data drive, each of the other data drives is processed separately, loading the respective 64 bytes of data into four registers (preserved across calls to the Parallel Lookup Multiplier). In addition, since the first check drive is the parity drive, its check data can be updated by directly adding these 64 bytes to it (using the Parallel Adder) before handling the non-parity check drives.

In step 550, the inner loop is called for the next data drive. In the inner loop (as with the first middle loop), each of the non-parity check drives is associated with a corresponding factor for the particular data drive. The factor is multiplied with each of the next data drive's data bytes using the Parallel Lookup Multiplier, and the results added to the check drive's check data.

Another such sequencer (Sequencer 2) can be used to reconstruct the lost data from the surviving data (using Algorithm 2). While the same column-by-column and row-by-row data access approaches are possible, as well as the same choices for handling the check drives, Algorithm 2 adds another dimension of complexity because of the four separate steps and whether to: (i) do the steps completely serially or (ii) do some of the steps concurrently on the same data. For example, step 1 (surviving check data generation) and step 4 (lost check data regeneration) can be done concurrently on the same data to reduce or minimize the number of surviving original data accesses from memory.

Empirical results show that method (2)(b)(ii), that is, row-by-row data access on all of the check drives and for both surviving check data generation and lost check data regeneration between data accesses performs best with the Parallel Lookup Multiplier when reconstructing lost data using Algorithm 2. Again, this may be due to the apparent minimization of the number of memory accesses (namely, one) of each chunk of surviving original data X to reconstruct the lost data and the absence of memory accesses of reconstructed lost original data Y when regenerating the lost check data. This embodiment of Sequencer 1 is described in more detail with reference to FIGS. 5-7.

FIGS. 5-7 show an exemplary method 600 for sequencing the Parallel Lookup Multiplier to perform the lost data reconstruction according to an embodiment of the present invention.

Referring to FIG. 5, in step 610, the Sequencer 2 is called. Sequencer 2 has many similarities with the embodiment of Sequencer 1 illustrated in FIG. 4. For instance, Sequencer 2 processes the data drive data in 64-byte chunks like Sequencer 1. Sequencer 2 is more complex, however, in that only some of the data drive data is surviving; the rest has to be reconstructed. In addition, lost check data needs to be regenerated. Like Sequencer 1, Sequencer 2 does these operations in such a way as to minimize memory accesses of the data drive data (by loading the data once and calling the Parallel Lookup Multiplier multiple times). Assume for ease of description that there is at least one surviving data drive; the case of no surviving data drives is handled a little differently, but not significantly different. In addition, recall from above that the driving formula behind data reconstruction is $Y=B^{-1} \times (W-A \times X)$, where Y is the lost original data, B^{-1} is the solution matrix, W is the surviving check data, A

22

is the partial check data encoding matrix (for the surviving check drives and the surviving data drives), and X is the surviving original data.

In step 620, the outer loop processes the next 64-byte chunk of data for each of the drives. Like Sequencer 1, the first surviving data drive is again handled specially since the partial check data $A \times X$ has to be initialized for each surviving check drive.

In step 630, the first middle loop is called, in which the partial check data $A \times X$ is initialized for each surviving check drive based on the first surviving data drive's 64 bytes of data. In this case, the Parallel Lookup Multiplier is called for each surviving check drive with the corresponding factor (from A) for the first surviving data drive.

In step 640, the second middle loop is called, in which the lost check data is initialized for each failed check drive. Using the same 64 bytes of the first surviving data drive (preserved across the calls to Parallel Lookup Multiplier in step 630), the Parallel Lookup Multiplier is again called, this time to initialize each of the failed check drive's check data to the corresponding component from the first surviving data drive. This completes the computations involving the first surviving data drive's 64 bytes of data, which were fetched with one access from main memory and preserved in the same four registers across steps 630 and 640.

Continuing with FIG. 6, in step 650, the third middle loop is called, which processes the other surviving data drives' corresponding 64-byte chunks of data. As with the first surviving data drive, each of the other surviving data drives is processed separately, loading the respective 64 bytes of data into four registers (preserved across calls to the Parallel Lookup Multiplier).

In step 660, the first inner loop is called, in which the partial check data $A \times X$ is updated for each surviving check drive based on the next surviving data drive's 64 bytes of data. In this case, the Parallel Lookup Multiplier is called for each surviving check drive with the corresponding factor (from A) for the next surviving data drive.

In step 670, the second inner loop is called, in which the lost check data is updated for each failed check drive. Using the same 64 bytes of the next surviving data drive (preserved across the calls to Parallel Lookup Multiplier in step 660), the Parallel Lookup Multiplier is again called, this time to update each of the failed check drive's check data by the corresponding component from the next surviving data drive. This completes the computations involving the next surviving data drive's 64 bytes of data, which were fetched with one access from main memory and preserved in the same four registers across steps 660 and 670.

Next, in step 680, the computation of the partial check data $A \times X$ is complete, so the surviving check data W is added to this result (recall that $W-A \times X$ is equivalent to $W+A \times X$ in binary Galois Field arithmetic). This is done by the fourth middle loop, which for each surviving check drive adds the corresponding 64-byte component of surviving check data W to the (surviving) partial check data $A \times X$ (using the Parallel Adder) to produce the (lost) partial check data $W-A \times X$.

Continuing with FIG. 7, in step 690, the fifth middle loop is called, which performs the two dimensional matrix multiplication $B^{-1} \times (W-A \times X)$ to produce the lost original data Y . The calculation is performed one row at a time, for a total of F rows, initializing the row to the first term of the corresponding linear combination of the solution matrix B^{-1} and the lost partial check data $W-A \times X$ (using the Parallel Lookup Multiplier).

US 10,666,296 B2

23

In step 700, the third inner loop is called, which completes the remaining $F-1$ terms of the corresponding linear combination (using the Parallel Lookup Multiplier on each term) from the fifth middle loop in step 690 and updates the running calculation (using the Parallel Adder) of the next row of $B^{-1} \times (W - A \times X)$. This completes the next row (and reconstructs the corresponding failed data drive's lost data) of lost original data Y , which can then be stored at an appropriate location.

In step 710, the fourth inner loop is called, in which the lost check data is updated for each failed check drive by the newly reconstructed lost data for the next failed data drive. Using the same 64 bytes of the next reconstructed lost data (preserved across calls to the Parallel Lookup Multiplier), the Parallel Lookup Multiplier is called to update each of the failed check drives' check data by the corresponding component from the next failed data drive. This completes the computations involving the next failed data drive's 64 bytes of reconstructed data, which were performed as soon as the data was reconstructed and without being stored and retrieved from main memory.

Finally, in step 720, the sixth middle loop is called. The lost check data has been regenerated, so in this step, the newly regenerated check data is stored at an appropriate location (if desired).

Aspects of the present invention can be also realized in other environments, such as two-byte quantities, each such two-byte quantity capable of taking on $2^{16}=65,536$ possible values, by using similar constructs (scaled accordingly) to those presented here. Such extensions would be readily apparent to one of ordinary skill in the art, so their details will be omitted for brevity of description.

Exemplary techniques and methods for doing the Galois field manipulation and other mathematics behind RAID error correcting codes are described in Appendix A, which contains a paper "Information Dispersal Matrices for RAID Error Correcting Codes" prepared for the present application.

Multi-Core Considerations

What follows is an exemplary embodiment for optimizing or improving the performance of multi-core architecture systems when implementing the described erasure coding system routines. In multi-core architecture systems, each processor die is divided into multiple CPU cores, each with their own local caches, together with a memory (bus) interface and possible on-die cache to interface with a shared memory with other processor dies.

FIG. 8 illustrates a multi-core architecture system 100 having two processor dies 110 (namely, Die 0 and Die 1).

Referring to FIG. 8, each die 110 includes four central processing units (CPUs or cores) 120 each having a local level 1 (L1) cache. Each core 120 may have separate functional units, for example, an x86 execution unit (for traditional instructions) and a SSE execution unit (for software designed for the newer SSE instruction set). An example application of these function units is that the x86 execution unit can be used for the RAID control logic software while the SSE execution unit can be used for the GF operation software. Each die 110 also has a level 2 (L2) cache/memory bus interface 130 shared between the four cores 120. Main memory 140, in turn, is shared between the two dies 110, and is connected to the input/output (I/O) controllers 150 that access external devices such as disk drives or other non-volatile storage devices via interfaces such as Peripheral Component Interconnect (PCI).

Redundant array of independent disks (RAID) controller processing can be described as a series of states or functions.

24

These states may include: (1) Command Processing, to validate and schedule a host request (for example, to load or store data from disk storage); (2) Command Translation and Submission, to translate the host request into multiple disk requests and to pass the requests to the physical disks; (3) Error Correction, to generate check data and reconstruct lost data when some disks are not functioning correctly; and (4) Request Completion, to move data from internal buffers to requestor buffers. Note that the final state, Request Completion, may only be needed for a RAID controller that supports caching, and can be avoided in a cacheless design.

Parallelism is achieved in the embodiment of FIG. 8 by assigning different cores 120 to different tasks. For example, some of the cores 120 can be "command cores," that is, assigned to the I/O operations, which includes reading and storing the data and check bytes to and from memory 140 and the disk drives via the I/O interface 150. Others of the cores 120 can be "data cores," and assigned to the GF operations, that is, generating the check data from the original data, reconstructing the lost data from the surviving data, etc., including the Parallel Lookup Multiplier and the sequencers described above. For example, in exemplary embodiments, a scheduler can be used to divide the original data D into corresponding portions of each block, which can then be processed independently by different cores 120 for applications such as check data generation and lost data reconstruction.

One of the benefits of this data core/command core subdivision of processing is ensuring that different code will be executed in different cores 120 (that is, command code in command cores, and data code in data cores). This improves the performance of the associated L1 cache in each core 120, and avoids the "pollution" of these caches with code that is less frequently executed. In addition, empirical results show that the dies 110 perform best when only one core 120 on each die 110 does the GF operations (i.e., Sequencer 1 or Sequencer 2, with corresponding calls to Parallel Lookup Multiplier) and the other cores 120 do the I/O operations. This helps localize the Parallel Lookup Multiplier code and associated data to a single core 120 and not compete with other cores 120, while allowing the other cores 120 to keep the data moving between memory 140 and the disk drives via the I/O interface 150.

Embodiments of the present invention yield scalable, high performance RAID systems capable of outperforming other systems, and at much lower cost, due to the use of high volume commodity components that are leveraged to achieve the result. This combination can be achieved by utilizing the mathematical techniques and code optimizations described elsewhere in this application with careful placement of the resulting code on specific processing cores. Embodiments can also be implemented on fewer resources, such as single-core dies and/or single-die systems, with decreased parallelism and performance optimization.

The process of subdividing and assigning individual cores 120 and/or dies 110 to inherently parallelizable tasks will result in a performance benefit. For example, on a Linux system, software may be organized into "threads," and threads may be assigned to specific CPUs and memory systems via the `kthread_bind` function when the thread is created. Creating separate threads to process the GF arithmetic allows parallel computations to take place, which multiplies the performance of the system.

Further, creating multiple threads for command processing allows for fully overlapped execution of the command processing states. One way to accomplish this is to number each command, then use the arithmetic MOD function (%) in

US 10,666,296 B2

25

C language) to choose a separate thread for each command. Another technique is to subdivide the data processing portion of each command into multiple components, and assign each component to a separate thread.

FIG. 9 shows an exemplary disk drive configuration 200 according to an embodiment of the present invention.

Referring to FIG. 9, eight disks are shown, though this number can vary in other embodiments. The disks are divided into three types: data drives 210, parity drive 220, and check drives 230. The eight disks break down as three data drives 210, one parity drive 220, and four check drives 230 in the embodiment of FIG. 9.

Each of the data drives 210 is used to hold a portion of data. The data is distributed uniformly across the data drives 210 in stripes, such as 192 KB stripes. For example, the data for an application can be broken up into stripes of 192 KB, and each of the stripes in turn broken up into three 64 KB blocks, each of the three blocks being written to a different one of the three data drives 210.

The parity drive 220 is a special type of check drive in that the encoding of its data is a simple summation (recall that this is exclusive OR in binary GF arithmetic) of the corresponding bytes of each of the three data drives 210. That is, check data generation (Sequencer 1) or regeneration (Sequencer 2) can be performed for the parity drive 220 using the Parallel Adder (and not the Parallel Lookup Multiplier). Accordingly, the check data for the parity drive 220 is relatively straightforward to build. Likewise, when one of the data drives 210 no longer functions correctly, the parity drive 220 can be used to reconstruct the lost data by adding (same as subtracting in binary GF arithmetic) the corresponding bytes from each of the two remaining data drives 210. Thus, a single drive failure of one of the data drives 210 is very straightforward to handle when the parity drive 220 is available (no Parallel Lookup Multiplier). Accordingly, the parity drive 220 can replace much of the GF multiplication operations with GF addition for both check data generation and lost data reconstruction.

Each of the check drives 230 contains a linear combination of the corresponding bytes of each of the data drives 210. The linear combination is different for each check drive 230, but in general is represented by a summation of different multiples of each of the corresponding bytes of the data drives 210 (again, all arithmetic being GF arithmetic). For example, for the first check drive 230, each of the bytes of the first data drive 210 could be multiplied by 4, each of the bytes of the second data drive 210 by 3, and each of the bytes of the third data drive 210 by 6, then the corresponding products for each of the corresponding bytes could be added to produce the first check drive data. Similar linear combinations could be used to produce the check drive data for the other check drives 230. The specifics of which multiples for which check drive are explained in Appendix A.

With the addition of the parity drive 220 and check drives 230, eight drives are used in the RAID system 200 of FIG. 9. Accordingly, each 192 KB of original data is stored as 512 KB (i.e., eight blocks of 64 KB) of (original plus check) data. Such a system 200, however, is capable of recovering all of the original data provided any three of these eight drives survive.

That is, the system 200 can withstand a concurrent failure of up to any five drives and still preserve all of the original data.

Exemplary Routines to Implement an Embodiment

The error correcting code (ECC) portion of an exemplary embodiment of the present invention may be written in software as, for example, four functions, which could be

26

named as ECCInitialize, ECCSolve, ECCGenerate, and ECCRegenerate. The main functions that perform work are ECCGenerate and ECCRegenerate. ECCGenerate generates check codes for data that are used to recover data when a drive suffers an outage (that is, ECCGenerate generates the check data J from the original data D using Sequencer 1). ECCRegenerate uses these check codes and the remaining data to recover data after such an outage (that is, ECCRegenerate uses the surviving check data W, the surviving original data X, and Sequencer 2 to reconstruct the lost original data Y while also regenerating any of the lost check data). Prior to calling either of these functions, ECCSolve is called to compute the constants used for a particular configuration of data drives, check drives, and failed drives (for example, ECCSolve builds the solution matrix B^{-1} together with the lists of surviving and failed data and check drives). Prior to calling ECCSolve, ECCInitialize is called to generate constant tables used by all of the other functions (for example, ECCInitialize builds the IDM E and the two lookup tables for the Parallel Lookup Multiplier).

ECCInitialize

The function ECCInitialize creates constant tables that are used by all subsequent functions. It is called once at program initialization time. By copying or precomputing these values up front, these constant tables can be used to replace more time-consuming operations with simple table look-ups (such as for the Parallel Lookup Multiplier). For example, four tables useful for speeding up the GF arithmetic include:

1. mvct—an array of constants used to perform GF multiplication with the PSHUFB instruction that operates on SSE registers (that is, the Parallel Lookup Multiplier).
2. mast—contains the master encoding matrix S (or the Information Dispersal Matrix (IDM) E, as described in Appendix A), or at least the nontrivial portion, such as the check drive encoding matrix H
3. mul_tab—contains the results of all possible GF multiplication operations of any two operands (for example, $256 \times 256 = 65,536$ bytes for all of the possible products of two different one-byte quantities)
4. div_tab—contains the results of all possible GF division operations of any two operands (can be similar in size to mul_tab)

ECC Solve

The function ECCSolve creates constant tables that are used to compute a solution for a particular configuration of data drives, check drives, and failed drives. It is called prior to using the functions ECCGenerate or ECCRegenerate. It allows the user to identify a particular case of failure by describing the logical configuration of data drives, check drives, and failed drives. It returns the constants, tables, and lists used to either generate check codes or regenerate data. For example, it can return the matrix B that needs to be inverted as well as the inverted matrix B^{-1} (i.e., the solution matrix).

ECCGenerate

The function ECCGenerate is used to generate check codes (that is, the check data matrix J) for a particular configuration of data drives and check drives, using Sequencer 1 and the Parallel Lookup Multiplier as described above. Prior to calling ECCGenerate, ECCSolve is called to compute the appropriate constants for the particular configuration of data drives and check drives, as well as the solution matrix B^{-1} .

ECCRegenerate

The function ECCRegenerate is used to regenerate data vectors and check code vectors for a particular configuration of data drives and check drives (that is, reconstructing the

original data matrix D from the surviving data matrix X and the surviving check matrix W, as well as regenerating the lost check data from the restored original data), this time using Sequencer 2 and the Parallel Lookup Multiplier as described above. Prior to calling ECCRegenerate, ECCSolve is called to compute the appropriate constants for the particular configuration of data drives, check drives, and failed drives, as well as the solution matrix B⁻¹.

Exemplary Implementation Details

As discussed in Appendix A, there are two significant sources of computational overhead in erasure code processing (such as an erasure coding system used in RAID processing): the computation of the solution matrix B⁻¹ for a given failure scenario, and the byte-level processing of encoding the check data J and reconstructing the lost data after a lost packet (e.g., data drive failure). By reducing the solution matrix B⁻¹ to a matrix inversion of a F×F matrix, where F is the number of lost packets (e.g., failed drives), that portion of the computational overhead is for all intents and purposes negligible compared to the megabytes (MB), gigabytes (GB), and possibly terabytes (TB) of data that needs to be encoded into check data or reconstructed from the surviving original and check data. Accordingly, the remainder of this section will be devoted to the byte-level encoding and regenerating processing.

As already mentioned, certain practical simplifications can be assumed for most implementations. By using a Galois field of 256 entries, byte-level processing can be used for all of the GF arithmetic. Using the master encoding matrix S described in Appendix A, any combination of up to 127 data drives, 1 parity drive, and 128 check drives can be supported with such a Galois field. While, in general, any combination of data drives and check drives that adds up to 256 total drives is possible, not all combinations provide a parity drive when computed directly. Using the master encoding matrix S, on the other hand, allows all such combinations (including a parity drive) to be built (or simply indexed) from the same such matrix. That is, the appropriate sub-matrix (including the parity drive) can be used for configurations of less than the maximum number of drives.

In addition, using the master encoding matrix S permits further data drives and/or check drives can be added without requiring the recomputing of the IDM E (unlike other proposed solutions, which recompute E for every change of N or M). Rather, additional indexing of rows and/or columns of the master encoding matrix S will suffice. As discussed above, the use of the parity drive can eliminate or significantly reduce the somewhat complex GF multiplication operations associated with the other check drives and replaces them with simple GF addition (bitwise exclusive OR in binary Galois fields) operations. It should be noted that master encoding matrices with the above properties are possible for any power-of-two number of drives 2^P=N_{max}+M_{max} where the maximum number of data drives N_{max} is one less than a power of two (e.g., N_{max}=127 or 63) and the maximum number of check drives M_{max} (including the parity drive) is 2^P-N_{max}.

As discussed earlier, in an exemplary embodiment of the present invention, a modern x86 architecture is used (being readily available and inexpensive). In particular, this architecture supports 16 XMM registers and the SSE instructions. Each XMM register is 128 bits and is available for special purpose processing with the SSE instructions. Each of these XMM registers holds 16 bytes (8-bit), so four such registers can be used to store 64 bytes of data. Thus, by using SSE

instructions (some of which work on different operand sizes, for example, treating each of the XMM registers as containing 16 one-byte operands), 64 bytes of data can be operated at a time using four consecutive SSE instructions (e.g., fetching from memory, storing into memory, zeroing, adding, multiplying), the remaining registers being used for intermediate results and temporary storage. With such an architecture, several routines are useful for optimizing the byte-level performance, including the Parallel Lookup Multiplier, Sequencer 1, and Sequencer 2 discussed above.

While the above description contains many specific embodiments of the invention, these should not be construed as limitations on the scope of the invention, but rather as examples of specific embodiments thereof. Accordingly, the scope of the invention should be determined not by the embodiments illustrated, but by the appended claims and their equivalents.

Glossary of Some Variables

A	encoding matrix (F × K), sub-matrix of T
B	encoding matrix (F × F), sub-matrix of T
B ⁻¹	solution matrix (F × F)
C	encoded data matrix ((N + M) × L) = $\begin{bmatrix} D \\ J \end{bmatrix}$
C'	surviving encoded data matrix (N × L) = $\begin{bmatrix} X \\ W \end{bmatrix}$
D	original data matrix (N × L)
D'	permuted original data matrix (N × L) = $\begin{bmatrix} X \\ Y \end{bmatrix}$
E	information dispersal matrix (IDM)((N + M) × N) = $\begin{bmatrix} I_N \\ H \end{bmatrix}$
F	number of failed data drives
G	number of failed check drives
H	check drive encoding matrix (M × N)
I	identity matrix (I _K = K × K identity matrix, I _N = N × N identity matrix)
J	encoded check data matrix (M × L)
K	number of surviving data drives = N - F
L	data block size (elements or bytes)
M	number of check drives
M _{max}	maximum value of M
N	number of data drives
N _{max}	maximum value of N
O	zero matrix (K × F), sub-matrix of T
S	master encoding matrix ((M _{max} + N _{max}) × N _{max})
T	transformed IDM (N × N) = $\begin{bmatrix} I_K & O \\ A & B \end{bmatrix}$
W	surviving check data matrix (F × L)
X	surviving original data matrix (K × L)
Y	lost original data matrix (F × L)

What is claimed is:

1. An accelerated error-correcting code (ECC) system operating across multiple drives, comprising:
 - at least one processing circuit comprising a plurality of central processing unit (CPU) cores that executes CPU instructions and loads original data from a main memory and stores check data to the main memory, each of the CPU cores comprising at least 16 registers, and each of the registers storing at least 8 bytes;

US 10,666,296 B2

29

at least one system drive comprising at least one non-volatile storage medium that stores the CPU instructions;

a plurality of data drives each comprising at least one non-volatile storage medium that stores at least one block of the original data;

at least four check drives each comprising at least one non-volatile storage medium that stores at least one block of the check data corresponding to the at least one block of the original data; and

at least one input/output (I/O) controller that receives the at least one block of the original data from a transmitter and that stores the at least one block of the original data to a main memory;

wherein the processing circuit, the CPU instructions, the main memory, the plurality of data drives, the at least four check drives, and the at least one I/O controller are configured to implement a multi-core erasure encoding system comprising:

original data in the main memory comprised of the at least one block of original data from the plurality of data drives;

check data in the main memory comprised of the at least one block of check data;

an encoding matrix for holding first factors in the main memory, the first factors being for encoding the original data in the main memory into the check data in the main memory; and

a scheduler for generating ECC data in parallel across a plurality of threads by:

dividing the original data in the main memory into a plurality of data matrices;

dividing the check data in the main memory into a plurality of check matrices;

assigning corresponding ones of the data matrices and the check matrices in the main memory to the plurality of threads, wherein each thread comprises an encoder, the encoder comprising at least a portion of the encoding matrix, a Galois Field (GF) multiplier, a Galois Field (GF) adder, and a sequencer for ordering operations through at least one of the data matrices, corresponding ones of the check matrices, and the at least a portion of the encoding matrix in the main memory using the GF multiplier and the GF adder to generate the check data in the main memory; and assigning the plurality of threads to the plurality of CPU cores of the processing circuit to concurrently generate the check matrices in the main memory from corresponding ones of the data matrices in the main memory.

2. The system of claim 1, wherein the scheduler divides the original data in the main memory and the check data in the main memory into a plurality of stripes, each of the plurality of stripes comprising at least:

one block of the original data; and

one corresponding block of the check data.

3. The system of claim 2, wherein the scheduler assigns the stripes to the plurality of threads such that, for each stripe of the plurality of stripes, the check data of the stripe is computed by no more than one of the plurality of threads.

4. The system of claim 3, wherein each of the plurality of threads corresponding to at least one of the plurality of stripes is assigned to a respective one of the plurality of CPU cores of the processing circuit.

5. An accelerated error-correcting code (ECC) decoding system operating across multiple drives, comprising:

30

at least one processing circuit comprising a plurality of central processing unit (CPU) cores that executes CPU instructions and loads original data and check data from a main memory and stores decoded check data corresponding to lost original data to the main memory, each of the CPU cores comprising at least 16 registers, and each of the registers storing at least 8 bytes;

at least one system drive comprising at least one non-volatile storage medium that stores the CPU instructions;

a plurality of data drives each comprising at least one non-volatile storage medium that stores at least one block of the original data;

at least four check drives each comprising at least one non-volatile storage medium that stores at least one block of the check data corresponding to the at least one block of the original data; and

at least one input/output (I/O) controller that receives the at least one block of the original data from a transmitter and that stores the at least one block of the original data to a main memory;

wherein the processing circuit, the CPU instructions, the main memory, the plurality of data drives, the at least four check drives, and the at least one I/O controller are configured to implement a multi-core erasure decoding system comprising:

original data in the main memory comprised of the at least one block of original data from the plurality of data drives;

check data in the main memory comprised of the at least one block of check data from the at least four check drives;

a solution matrix, the solution matrix comprising factors for decoding the check data in the main memory to reproduce lost original data in the main memory; and

a scheduler for decoding ECC data in parallel across a plurality of threads by:

dividing the original data in the main memory into a plurality of data matrices;

dividing the check data in the main memory into a plurality of check matrices;

assigning corresponding ones of the data matrices and the check matrices in the main memory to the plurality of threads, wherein each thread comprises a decoder, the decoder comprising at least a portion of the solution matrix, a Galois Field (GF) multiplier, a Galois Field (GF) adder, and a sequencer for ordering operations through at least one of the data matrices, corresponding ones of the check matrices, and the at least a portion of the solution matrix in the main memory using the GF multiplier and the GF adder to decode the check data in the main memory into lost original data in the main memory; and

assigning the plurality of threads to the plurality of CPU cores of the processing circuit to concurrently regenerate portions of the data matrices corresponding to lost original data in the main memory from corresponding ones of the check matrices in the main memory.

6. The system of claim 5, wherein the scheduler divides the original data in the main memory and the check data in the main memory into a plurality of stripes, each of the plurality of stripes comprising at least:

one block of the original data; and

one corresponding block of the check data.

US 10,666,296 B2

31

7. The system of claim 6, wherein the scheduler assigns the stripes to the plurality of threads such that, for each stripe of the plurality of stripes, the decoding of the check data of the stripe corresponding to the lost original data is computed by no more than one of the plurality of threads.

5

8. The system of claim 7, wherein each of the plurality of threads corresponding to at least one of the plurality of stripes is assigned to a respective one of the plurality of CPU cores of the processing circuit.

10

* * * * *

32

EXHIBIT I



TANTALO & ADLER LLP

Michael S. Adler
Direct: (310) 734-8694
E-mail: madler@ta-llp.com

July 5, 2013

VIA U.S. MAIL, FAX AND EMAIL/PDF

Legal Department/Intellectual Property
USENIX
2560 Ninth Street, Suite 215
Berkley, CA 94710
United States
Email: office@usenix.org
Fax: 510-548-5738

Re: *Improper Disclosure of StreamScale Intellectual Property*

Dear USENIX:

Our firm is IP litigation counsel for StreamScale, Inc., an industry leader in intellectual property relating to storage systems, including RAID drives. StreamScale has some of the most advanced RAID systems in the marketplace, far faster and more reliable than any other competing system.

We are writing because it recently came to our attention that your organization has published at least three different papers that were authored in significant part by Professor James S. Plank of the EECS Department of the University of Tennessee. These papers include "Screaming Fast Galois Field Arithmetic Using Intel SIMD Instructions," "Erasure Coding for Storage Applications," and "SD Codes: Erasure Codes Designed for How Storage Systems Really Fail." We understand that all three of these papers were presented at this year's FAST13 conference.

The subject of these three papers inappropriately revealed confidential information from StreamScale which is the subject of at least two pending patent applications. The first of these patent applications was filed in 2011 and yesterday became public, with the title "ACCELERATED ERASURE CODING SYSTEM AND METHOD" (U.S. Patent Application No. 20130173996). The second is a continuation of the first and likewise yesterday became public, with the title "USING PARITY DATA FOR CONCURRENT DATA AUTHENTICATION, CORRECTION, COMPRESSION, AND ENCRYPTION" (U.S. Patent Application No. 20130173956).

In publishing and speaking about the three papers identified above, Professor Plank discussed the fact that the key technique of using SIMD instructions at the heart of his papers was a closely-guarded "secret handshake" that he had allegedly figured out on his own. In fact, Professor Plank knew it was closely-guarded information because he had learned it from StreamScale pursuant to an express written confidentiality agreement.

TANTALO & ADLER LLP

June 13, 2013
Page 2

In this regard, you should be aware of the following facts:

- 1) On or about August 3, 2011, Professor Plank became a paid consultant for StreamScale. As an essential element of his consulting agreement with StreamScale, Professor Plank agreed to maintain all information received from StreamScale (and information derived therefrom) confidential.
- 2) Pursuant to the Consulting Agreement (and the confidentiality clauses therein), StreamScale disclosed to Professor Plank information that made clear StreamScale's proprietary method of accelerating Galois Field processing using SIMD instructions. Generally speaking, StreamScale disclosed to Professor Plank detailed performance measurements of the StreamScale proprietary method of accelerating Erasure Coding, and that StreamScale used a process that could multiply 64 bytes of data in 41 instructions, from which it would have been (and was) clear that the StreamScale method used SIMD instructions. Moreover, among other things, StreamScale specifically disclosed to Professor Plank on or about August 5, 2011 that StreamScale used the PSHUFB Intel SSE instruction combined with two tables of 8 bit constants, as specifically applied to achieve high performance Galois Field multiply operations for Reed Solomon codes.
- 3) In violation of his consulting agreement, it appears that Professor Plank shared this information with his co-authors and subsequently included the confidential process details in the public papers. For example, in the first paper cited above, this process (the PSHUFB instruction combined with two tables of 8 bit constants) was identified as the "real enabling SIMD instruction for Galois Fields". That is precisely what StreamScale disclosed to Professor Plank in August of 2011.
- 4) In addition, you should be aware that StreamScale itself submitted at least one proposed paper for FAST13 which revealed that StreamScale had achieved the exceptional level of performance that Professor Plank claimed to identify on his own. That paper was titled "An Erasure Coding Performance Metric for Windows 8," and a copy of that paper is attached hereto. In other words, FAST and its committees were or should have been aware that Professor Plank's "discovery" was not unique and that at least one company had commercialized those techniques. Under these circumstances, we are disappointed that USENIX did not question Professor Plank more closely about the relationship between his "work" and StreamScale's results.

StreamScale is writing with several goals in mind.

TANTALO & ADLER LLP

June 13, 2013

Page 3

First, to the extent that Professor Plank has repeatedly (and improperly) claimed that his techniques are “open source” and exist free of any patent protection, you and your members should be aware that the techniques actually belong to StreamScale and are in fact the subject of at least two pending patent applications. While the patents have not yet issued, StreamScale is confident that they will issue in time. Moreover, the techniques were created originally by StreamScale and – to the extent that Professor Plank added anything to such techniques – his agreement with StreamScale provides for StreamScale ownership of any improvements based on StreamScale’s property. StreamScale believes that your members should be aware that Professor Plank’s statements about the “open source” nature of these techniques are inaccurate and that they use such techniques at their own risk. We therefore believe that it is appropriate for Usenix to inform all of its members ASAP of the fact that the techniques at issue are subject to StreamScale’s pending patent claims.

Second, it appears likely that StreamScale will be forced into litigation with Professor Plank (and possibly others) over the inappropriate disclosure of StreamScale’s intellectual property. You and your members should all be aware of your obligations at the very least to retain any and all communications with or involving Professor Plank that involve or relate to the use of SIMD instructions for Galois Field multipliers. In addition, to the extent that USENIX is merely an innocent third party without any agenda (as we hope is the case), we would ask that USENIX, its FAST committee members, and Professor Plank’s co-authors please voluntarily provide to StreamScale any and all communications with or regarding Professor Plank since August 1, 2011 as well as any and all communications regarding Professor Plank’s three papers at this year’s FAST13 conference or StreamScale’s two proposed papers.

Third, we must insist that USENIX immediately cease publishing/displaying the papers and any related material (such as audio and/or video material) by Professor Plank that improperly disclose confidential StreamScale information (and which improperly claim that such information is “open source” and not subject to patent protection). As you can readily see by reviewing the public patent applications, the technology identified by Professor Plank and his various co-editors is *not* open source or free of patent protection.

In connection with our request that USENIX withdraw the papers, you should be aware that StreamScale expects to issue a press release soon to caution the public against inappropriate use of the StreamScale technology. We are enclosing a draft of that press release. We would like to be able to indicate in the press release that USENIX has taken the proper steps of withdrawing the papers in light of the information, so please let us know if you expect to do so.

StreamScale hopes that USENIX and its committee members were truly ignorant of Dr. Plank’s violation of his confidentiality obligations and that USENIX did not realize the extent to which its publications and conference were being used to inappropriately publish confidential intellectual property (and mislead the industry about the status of the same). To the extent that USENIX has been merely an innocent third-

TANTALO & ADLER LLP

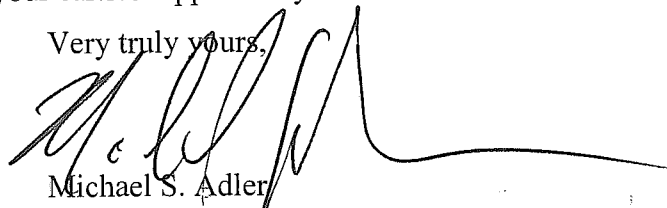
June 13, 2013

Page 4

party misused by Professor Plank, we would hope that USENIX will work to rectify the harm that he has caused to StreamScale (and potentially to the various USENIX members) by virtue of his misuse of USENIX and FAST.

We look forward to hearing from you at your earliest opportunity.

Very truly yours,

A handwritten signature in black ink, appearing to read 'Michael S. Adler', with a long horizontal flourish extending to the right.

Michael S. Adler



RELEASED FOR PUBLICATION **DRAFT**

StreamScale Provides Notice of Ownership of Fastest Erasure Code Technology Disclosed at FAST2013

*Technology Wrongly Identified at FAST as "Open Source:"
Technology Only Available for Use Through Properly Executed Licensing
Agreements With StreamScale*

Los Angeles, CA – July 15, 2013: StreamScale, a leading developer providing technology to protect storage systems from data loss and corruption, announced today it has discovered publication of alleged "open source" materials that include the Company's confidential and patent-pending technology. Such technology is not "open source" and was improperly disclosed in publications and talks at FAST13.

In particular, StreamScale has learned that one of its consultants was an author of a paper entitled "Screaming Fast Galois Field Arithmetic Using Intel SIMD Instructions," which contains technology StreamScale shared with that consultant under a confidential relationship. The technology is protected intellectual property owned by StreamScale, and it is not "open source." StreamScale announced it will enforce its rights to this technology and parties may not use this technology in any way without a proper license from StreamScale.

"We have recently become aware that StreamScale confidential information and trade secrets were disclosed in a paper delivered at FAST2013 without StreamScale's approval or knowledge," said Michael H. Anderson, president and CEO, StreamScale. "A paid consultant revealed information provided to him by StreamScale in violation of the terms of his contract. It is clear that the research and resultant findings disclosed by the consultant and his co-authors were a direct result of StreamScale's proprietary IP. StreamScale's technologies are not only the subject of trade secret protection but are also subject to copyright protection and at least two pending patent applications."

Continued Anderson, "Do not be misled by claims that the technology is "open source" and do not assume that such information downloaded from USENIX or University websites is unprotected. We have asked these organizations to remove the content from their websites and conference proceedings. I am informing any company or person that uses this protected technology without license from StreamScale does so at their own risk."

Further investigation has discovered additional papers have been published including the patent pending technology and must not be used without license from StreamScale. While it is unknown how many papers have included the information, here are several that people should understand include StreamScale's protected information:

- M. Blaum and J. S. Plank, Construction of two SD Codes, arXiv: 1305.1221, May, 2013.
- J. S. Plank and M. Blaum, "Sector-Disk (SD) Erasure Codes for Mixed Failure Modes in RAID Systems," Technical Report CS-13-708, University of Tennessee EECS Department, May, 2013.
- J. S. Plank, "Open Source Encoder and Decoder for SD Erasure Codes - Revision 2.0," Technical Report CS-13-707, University of Tennessee EECS Department, May, 2013.
- J. S. Plank, "Open Source Encoder and Decoder for SD Erasure Codes," Technical Report CS-13-704, University of Tennessee EECS Department, January, 2013.
- J. S. Plank, E. L. Miller and W. B. Houston, "GF-Complete: A Comprehensive Open Source Library for Galois Field Arithmetic, Revision 0.1," Technical Report CS-13-703, University of Tennessee EECS Department, January, 2013.

At the core of the StreamScale technology is high performance erasure coding that optimizes Galois Field arithmetic (GF). StreamScale discovered how to utilize specific instructions in the Intel SSE3 SIMD instruction set to perform GF arithmetic so that it is only limited by the L2/L3 cache, providing up to a 12x improvement over the best performing existing XOR codes. As reflected in patent filings dating back to 2011, implementing erasure coding with StreamScale technology enables the creation of the best performing most reliable storage systems that eliminate data loss and corruption due to disk failure, service errors, silent data corruption, and unrecoverable read errors while increasing system performance.

"Unfortunately we have read comments online that individuals and companies are using these protected technologies based on the FAST2013 paper to optimize storage systems and other products without license from StreamScale," concluded Anderson. "We have the utmost respect for USENIX and technology companies in the industry and anticipate they will do the right thing by removing the improperly disclosed information from their websites and conference proceedings and seek a license agreement from us or remove the StreamScale IP for their solutions."

About StreamScale

StreamScale leads the industry in providing technology to protect storage systems from data loss and corruption. The Big Parity® Verified Erasure Coding® system is available to all data storage manufacturers and system integrators. By including Big Parity in their storage system manufactures can achieve 10 orders of magnitude better RAID reliability and up to 30X faster system performance.

Press Contacts



Curtis Chan
COGNITIVE IMPACT
Office: +1 714.447.4993
Fax: +1 714.447.6020
E-mail: curtis@cognitiveimpact.com

An Erasure Coding Performance Metric for Windows 8

Abstract

Electronic data itself may be ethereal, but the devices that store it and transport it are physical. As such, when electronic data travels in space and time over physical devices, it is subject to noise and deterioration. The standard solution to eliminate this noise is Reed-Solomon coding, typically implemented in hardware and used today in every data storage device. We present a measurement of Reed-Solomon coding in a new application: a Windows 8 storage device driver. This software application brings the benefits of noise elimination much closer to the producer or consumer of the data, and has surprisingly good performance, even for very strong encoding. Our results challenge the common assumption that standard Reed-Solomon codes, realized with Galois Field multiply operations, are too computationally expensive to be effectively deployed in software. We also include a brief general discussion of Reed-Solomon codes and of the computational costs of ECC.

1 Introduction

Reed-Solomon codes are used throughout the communication and storage industries to protect data from noise. All non-volatile storage devices, most server DRAMs and all disk arrays use Reed-Solomon or Reed-Solomon-like codes to detect and correct data errors. These codes are often called Erasure Codes or Error Correcting Codes (ECC), both of which have the same mathematical basis.

Noise that affects electronic data has many sources, including cosmic rays, reported and unreported device or communication failures, human error, power problems, physical media degradation and many others. As distance, time, and data volumes increase, so does this noise and deterioration. Unless a methodology is applied to compensate for this noise, electronic data will be lost in the noise, rendering it useless.

To understand the noise cancellation and regenerative effects of ECC in real applications, we suggest that it is important to consider two separate metrics. The first metric is the distance in space and time between the creation of the data and the protection of the data. As long as the data remains unprotected, it can be corrupted in an undetectable manner. The second metric is the strength of the

codeword. The stronger the codeword, the more noise can be eliminated.

An ideal data communication or storage system would apply ECC protection as close as possible in space and time to the creator of the data, remove ECC protection as close as possible to the data consumer, and would support very strong codewords. This combination of features would best protect the data from the inevitable noise it encounters in its lifecycle.

For application data, the creator or consumer of the data is an application program, executing on an application processor. In the past, application processors have been too expensive to apply to anything but application execution. However, in modern processors with multiple cores, these resources are often underutilized. This is primarily because legacy applications were designed for a single processor, and most processors sold now are multi-core.

From a space and time perspective, a core in an application processor is an ideal location for ECC-based noise elimination. The distance in space and time between the application data creation and the ECC logic is nearly zero. The physical distance may actually be zero, since the same processor that the application used to generate the data and initiate the transfer could be used to encode or decode the ECC.

In Windows 8, we accomplished this near-zero distance noise elimination solution with a software device driver implemented within a Windows 8 device driver model called *StorPort virtual miniport* [7]. Our implementation is an autonomous piece of software apart from using the Microsoft driver framework. The driver acts on behalf of the application that produces the data, and encodes this data with ECC prior to storing it on a local or remote memory system. When an application reads the data in the future, this same driver applies ECC logic to eliminate any noise that was introduced as the data travelled through space and time.

For Windows 8 applications that use this ECC driver, the ECC encoding and decoding provides strong data protection and noise immunity. The ECC logic ensures that all data communication paths and memory devices between from the application processor through the storage are fully protected from both reported and unreported data errors, regardless of their source.

For example, if a cosmic ray from deep space caused a soft failure of the computer's DRAM, the ECC logic

would detect it and correct it, without requiring any additional hardware. Or, if a flash device media degraded to the point that a sector was unreadable, the driver would use ECC to recover the data. With a thoughtful arrangement of the codewords and devices, even full device failures can be considered “noise”, and data can be recovered in their absence. With strong codewords, multi-device failures could be tolerated without requiring immediate service. This is very strong application data protection, especially compared to typical Windows 8 desktop systems without any DRAM ECC.

This paper provides measurements of and discussions about such a Windows 8 ECC device driver as well as a discussion of the coding theory on which the driver is built. This driver implements ECC-protected memory that can be benchmarked with standard tools such as the ATTO Disk Benchmark and Intel’s IOMeter [1, 8]. By varying the strength of the ECC codewords and the number of cores used by the driver, we present a series of measurements that show Windows 8 can provide very high noise immunity for application data using standard Reed-Solomon codes based on Galois Field multiply operations.

2 Our Windows 8 Software ECC

In the current section we discuss a set of ideas and ECC technologies which we call jointly *Virtual ECC*. Subsequently, we introduce our software implementation of *Virtual ECC* for the new Windows 8 operating system from Microsoft.

The current section adequately describes our main result for those readers who are familiar with Reed-Solomon codes and their application to RAID. However, an interested reader will find the background material organized in several additional sections of this paper. Here, occasionally we make a (forward) reference to this material, hopefully with minimal distraction to an expert reader.

2.1 Virtual ECC

The concept of *Virtual ECC* is quite simple: we mean an approach to data protection in which extra bits are added to the data as early as possible after data inception, and stored as its integral part forever. The virtuality of this approach alludes to the fact that the manner in which these extra bits are chosen is not fixed by a single physical hardware implementation but may be adjusted to suit the varying needs of different applications. Strong encoding may be used in some cases, and weak encoding in others, all on the same hardware.

This contrasts with the typical hardware-based approach. Hardware-assisted ECC typically works as fol-

lows:

1. Dedicated hardware encodes the input data by adding a fixed number of redundant bits on the fly when the data reaches the hardware device.
2. The extra bits are removed when the data is shipped off, typically to another hardware device which in turn may add ECC bits of its own.
3. When the data travels between devices through some interconnect fabric, other bits, usually much weaker than ECC, may or may not be added or checked depending on the technology or vendor.

Thus, understanding the actual data protection as data moves through a system may be very complex. As a result, the probability of data loss is dominated by the weakest link, which likely will remain unknown until after a failure occurs.

By comparison, *Virtual ECC* deals with these unknowns in advance, with a scheme that matches the intended use and environment of the data. For data stored on very large systems, commonly called cloud infrastructures, very strong codewords can be applied to data, and then spread out between devices, device racks, facilities, even continents. For small systems, like home storage for multimedia libraries, weaker codewords would suffice.

Virtual ECC has the power to transform the lowest reliability devices into high reliability devices, and to eliminate another important problem that occurs frequently in large systems — silent data corruption. Since the protection bits are generated early in the life of the data, and stay attached to the data throughout its lifecycle, they guarantee the correctness of the data. Generating and checking ECC near the application eliminates all practical possibility of silent errors in any device that the data may encounter.

For example, here is a minimal list of devices that data will encounter on a typical computer: CPU, DRAM, PCI, Local Storage Controller, Remote Storage Controller, Flash, Tape or Disk media, and finally Interconnects (SATA, SAS, FC, IB). Strikingly, most computers use DRAM with no ECC protection. However, by using *Virtual ECC*, they are transformed into devices with much higher reliability than computers with hardware ECC. With *Virtual ECC*, data is protected from the moment the message is transformed into a codeword, and remains protected, with a known codeword strength, until it is used again. This protection covers every device the data encounters during its lifecycle. Corrupted codewords, even those silently corrupted, are easy to detect and correct regardless of the correctness of the devices used to store or transport the data.

We can also point to the following advantages of *Virtual ECC* over current ECC schemes:

1. Strong codes are millions of times more reliable for the same overhead or cost.
2. The ability to upgrade the reliability of existing hardware as well as future hardware at a software-only cost.
3. High configurability — ECC protection can be matched to the data and the environment.
4. Device lifetimes can be extended — ECC can be increased as devices deteriorate.

2.2 Our Virtual ECC Benchmarks

Virtual ECC is arguably the best solution for comprehensive data protection, and our implementation of it under Windows, in addition to the general benefits of *Virtual ECC*, offers additional advantages. They are rooted in the modern OS and hardware technology:

- A new, “software friendly” ECC algorithm.
- Leveraging of multi-core architecture.
- Faster processing than hardware-based alternatives.

We emphasize the light load on the CPU, which should become even lighter with new generations of processors. Thus, we have achieved unsurpassed flexibility without taxing hardware resources. The algorithm used by us and its computational complexity are covered in Section 4.

By writing the driver, we have demonstrated that the software implementation of ECC is *in practice* faster than a conceivable hardware implementation. Our ECC leverages the fastest, superbly tested, and most computationally advanced component of the system, the CPU ASIC. This is the most highly optimized part of a computer system and most capable for implementing advanced mathematical algorithms. Thus, by utilizing the CPU we are able to cash in on the tremendous resources which went into its development.

In contrast, the typical peripheral hardware design is subject to many compromises, and as a result ECC runs at a fraction of the speed possible with the CPU. By its nature, the CPU development enforces much higher quality assurance than a typical peripheral would.

Since in our *Virtual ECC* approach ECC is an integral part of the data, it can be easily implemented utilizing new infrastructures, based for instance on cloud storage. We may either replace or add to the existing ECC. Given that the reliability of the underlying storage system is available or can be estimated, we may estimate how much extra data protection we need, to balance the extra cost involved against our data protection need. RAID failure modeling may be used by the designer of the RAID system to determine the major parameters of the data protection, such as

the ratio of check to data drives. Further information on RAID failure modeling is provided in Section 5.

In Table 1 and Table 2 we present performance measurements of the StorPort virtual miniport driver on several hardware configurations. The measurements come from the industry-standard Disk Benchmark software from ATTO Technology, Inc [1]. We tested write and read operations with a varying number of bytes until sustained throughput was achieved. We also took steps to ensure that we measure the cost of ECC only, in isolation from other factors, such as the drive speed. This isolation was achieved by interposing a write-back cache in front of the real hard drive cache. The cache was sufficiently large to ensure the cache was not flushed before our measurements are performed. Although other costs are captured in this measurement, such as the cost of the DRAM to CPU data movement, they should be considered negligible as compared to the cost of performing ECC operations.

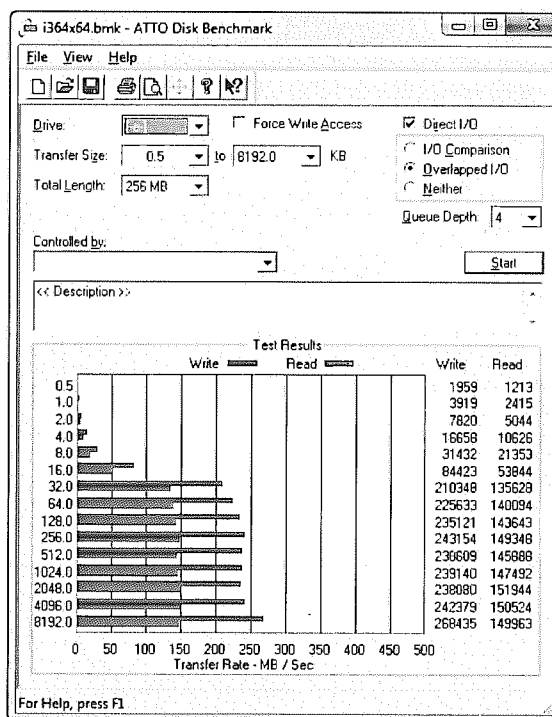


Figure 1: Screen capture of the GUI of the ATTO benchmarking software with sample measurements of the StorPort virtual miniport driver.

When configured with an extremely strong codeword, 64 data bytes + 64 check bytes, we achieved write speed exceeding 600MB/sec and read speed of over 350MB/sec. With a weaker codeword, 64 data bytes + 10 check bytes, the throughput is in excess of 3GB/sec. In all experiments,

we utilized 2–4 cores¹. As usual, the performance analysis of a real computer system can only be known approximately. Thus, the sustained throughput rate for 10 and 64 check drives do differ *roughly* by a factor of 5. From the point of view of reliability (see Section 5 for more details on this subject), the difference still may be justified in some demanding applications. The difference in codeword strength between 64 and 10 check bits is several orders of magnitude in probability of data loss, and may be well worth it when preservation of data is paramount. The reader may also note somewhat slower read performance, which is not accounted for by theoretical operation counts.

Table 1: The driver write/read performance: 64 data bytes, 64 check bytes.

Volume of Data	Hardware configuration				
	Celeron 1.6Ghz	Pentium 2.2Ghz	i3 2.4Ghz	i5 2.7Ghz	i7 2.2Ghz
Write Performance (transfer rate in kiB/sec)					
512b	1445	1955	1959	2740	2540
1k	2774	3957	3919	5562	5132
2k	5508	8070	7820	11377	10541
4k	11959	16181	16658	24035	22415
8k	24576	33557	31432	55158	51447
16k	59076	79897	84423	150226	144340
32k	116260	161729	210348	467021	453199
64k	121972	173399	225633	482527	495103
128k	125136	181594	235121	498789	496325
256k	130043	171095	243154	662886	597922
512k	125144	172442	238609	666092	721753
1024k	127522	178956	239140	664444	708103
2048k	129366	180024	238080	661171	669713
4096k	134892	216480	242379	651542	651542
8192k	125144	173557	268435	651542	614268
Read Performance (transfer rate in kiB/sec)					
512b	914	1270	1213	1798	1714
1k	1808	2528	2415	3648	3479
2k	3624	5190	5044	7492	7211
4k	7430	10702	10626	15906	15321
8k	15641	21609	21353	36883	35703
16k	37321	52461	53844	104162	104667
32k	76249	110461	135628	300452	296023
64k	80669	116260	140094	312125	312125
128k	81035	122819	143643	321900	325825
256k	90697	112649	149340	418961	367121
512k	81344	114227	145000	418496	448460
1024k	80249	116711	147492	420368	443171
2048k	85353	122016	151944	417566	426088
4096k	95520	112081	150524	414800	405841
8192k	82090	114961	149963	415718	387166

¹The precise count of resources such as cores and hardware threads is complicated by such technologies as hyperthreading, where some of the computational resources are shared; thus, we would rather think of the capabilities of a system as a whole than the particulars of the CPU architecture.

Table 2: The driver write/read performance: 64 data bytes, 10 check bytes.

Volume of Data	Hardware configuration				
	Celeron 1.6Ghz	Pentium 2.2Ghz	i3 2.4Ghz	i5 2.7Ghz	i7 2.2Ghz
Write Performance (transfer rate in kiB/sec)					
512b	6632	8832	6415	10544	9631
1k	13789	17408	12579	21089	18130
2k	27068	32768	26295	41867	37376
4k	47300	66228	53115	83284	77596
8k	93499	120356	110235	164526	169387
16k	189665	272531	268435	342879	376504
32k	356554	523182	648269	897754	930968
64k	438645	621217	754581	1576887	1823763
128k	487893	675162	891806	2274442	2191108
256k	507391	691666	995109	3200232	2796029
512k	511305	689904	1022611	3314017	3730102
1024k	522502	713324	1010580	3338749	3587675
2048k	522502	691519	984482	3277737	3587998
4096k	511305	703045	986895	3146250	3347075
8192k	510091	685102	1115746	3079309	3271974
Read Performance (transfer rate in kiB/sec)					
512b	5409	7132	5081	8151	6843
1k	11008	13721	10290	16262	13789
2k	21961	26800	20127	32524	28089
4k	38814	49192	41261	66774	58658
8k	78223	107531	83284	135251	138313
16k	159695	231204	210457	327419	350177
32k	308623	457708	496325	758978	1034229
64k	358454	515051	590595	1514190	1597065
128k	392254	555745	678416	1879048	1830975
256k	402653	565640	745924	2577017	2329764
512k	407602	588451	759722	2625285	3004874
1024k	418496	586388	747384	2664371	2960685
2048k	407602	580749	743931	2631720	2850055
4096k	399655	600974	745654	2591332	2763306
8192k	403229	560538	745385	2464543	2730981

2.3 Operation Count vs. Real Performance

Let us focus on the 64 data + 64 check bytes configuration, achieving write speed of 600MB/sec. Based on our knowledge of Reed-Solomon coding, we find that the number of Galois field operations required to generate a codeword from a message is equal to the number of check bytes per each byte of the input message. In Section 4 we provide the theoretical basis for this claim. Since in Reed-Solomon coding of RAID check bytes map 1:1 to check drives, this number is also equal to the number of check drives. This means that each of the 64 bytes of inputs contributes 64 Galois field operations to the cost of creating the 128 byte codeword. 600MB/s translates into 6×10^8 bytes of input per second, or $6 \times 10^8 \times 64 = 384 \times 10^8$ Galois operations per second. We feel that this is a very important metric of system performance and justifies giving it a name; we propose GalOPS (Galois Operations per Second). We may think of GalOPS as analogous to FLOPS which characterizes floating point performance.

In Section 4 we included further discussion of this metric.

In our example the performance of our driver suggests that the performance of our system is 38 GigaGalOPS. Based on the clock speed of 2.4GHz, we perform about 16 Galois field operations per clock cycle, or 4 Galois operations per clock cycle per processor, given a quadcore system. These estimates are imprecise, but it is certain that we must be able to perform *at least* 4 Galois operations in one clock cycle in each processor to justify the measured data. Using another method based on observing hardware performance counters, we indeed confirmed that this simple estimate of the actual performance is accurate (see Section 2.4). It is clear that the resulting performance is due to both multithreading and utilization of the pipelined architecture of the modern CPU.

When we use the codeword of 64 data bytes with 10 check bytes (still very strong protection), the I/O scales up by a factor of 5 but the number of Galois operations is 6.4 times smaller, resulting in approximately the same GalOPS measurement. This suggests that GalOPS indeed are a measure of the performance of the system, and not of the particulars of the algorithm used.

2.4 Measurement Methodology Details

For those readers who would like to more closely follow what our process, in this section we include additional details of our approach. They can be skipped without affecting the ability to understand the rest of the paper.

As we mentioned, Windows supplies a driver model *StorPort virtual miniport*, as well as examples to aid programmers who develop drivers [7]. The StorPort virtual miniport model itself was used to develop our *Virtual ECC* driver, but we did not use the Microsoft example code.

Inside this driver model, Windows also provides a set of measurement tools that can gather real time events at high speed and with low overhead. These tools are called *Event Tracing for Windows (ETW)*. We used these tools to gather real time performance information about the driver so we could understand how it behaved in a real operating system environment, and verify actual performance against modeled results.

Using these tools, we gathered detailed measurements of each IO that the benchmark generated as it was executed by the driver, and saved these measurements for later analysis. To bound the scope of our measurements, we focused on a particular processor and transfer size, though this same technique could have been applied to any processor or transfer size.

For each IO, we recorded the total execution time. In addition, we recorded three important events to understand performance at a more detailed level:

1. The time that the cache was searched and locked;

2. The time that the data was transferred to or from the benchmark to the cache;
3. The time it took to compute the ECC bytes for the request.

A typical 32k write request on an i7, for example, took 1 microsecond or less to search and lock the cache, about 6 microseconds to move the data from the ATTO benchmark memory to the driver cache memory, and about 400 microseconds to compute the ECC. A typical read request had a similar cache search and lock time, but a larger time (about 600 microseconds) to compute the ECC. The cause for this difference should be further investigated.

We included two charts, Figures 2 and 3, which show the ECC computation time in a 64 data bytes + 64 check bytes codeword configuration for a 32k IO. The first chart shows the time required to regenerate lost data using ECC, the second chart shows the time required to generate the parity bytes for a write. Though it is clear from these measurements that generating parity is faster than regenerating data, the cause should also be further investigated.

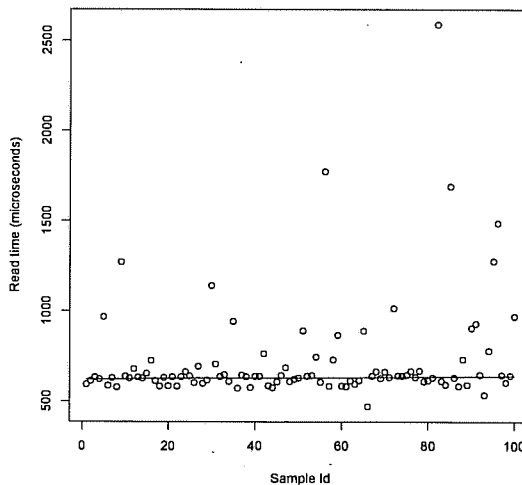


Figure 2: Measurements of the first 100 read operations @32k, 64+64 codeword. The line is smoothed data generated using the LOWESS method.

3 Review of Coding Theory

In this section we discuss the essentials of coding theory that allow for ECC in computing systems. Here, we will limit our discussion to linear block codes over finite fields focusing on Reed-Solomon-like codes in particular. These codes are particularly well suited for data protection in

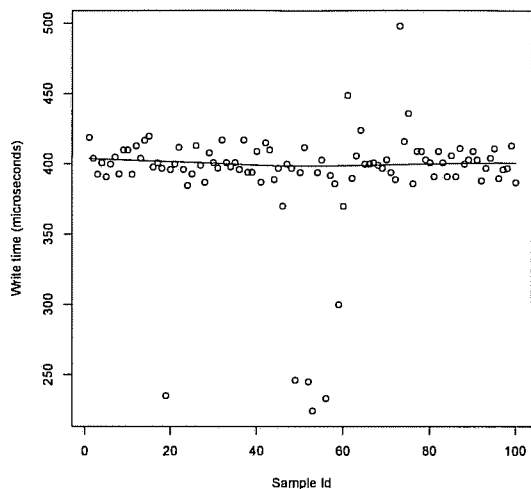


Figure 3: Measurements of the first 100 write operations @32k, 64+64 codeword. The line is smoothed data.

computer systems. A more detailed discussion of coding theory, linear block codes, and Reed Solomon codes can be found in standard texts on this subject [5, 12, 16].

In Reed-Solomon ECC, data originates as a string of bits which are first grouped into *characters* consisting of l bits of data; $l = 8$ is typical and corresponds to byte-sized characters. These characters are then grouped into k character *messages*. By adding an additional $m = n - k$ carefully chosen characters to a message, a group of n characters (the *codeword*) is created. Thus codewords are somewhat redundant, containing more characters than the message. The set of all possible codewords constitutes the *code*; this set is dependent on the manner in which the redundant characters are chosen.

The codeword is then transmitted or stored for later use and in the process is subjected to random errors. We refer to this codeword post storage or transmission as the *received word*. Due to the redundancy, the codeword and thus the original message may be recovered from the received word if the number of errors is not too great. The *rate* $R = \frac{k}{n}$ of the code measures the degree of redundancy in the codeword. $R = 1$ indicates there is no redundancy and no error correction capacity, and R near zero indicates there is a lot of redundancy and thus error correction capacity, but little information content.

3.1 Data Encoding

The essence of the problem of ECC is how to choose the redundant characters such that a message may be recovered from a received word in the presence of a large number of errors in a computationally efficient manner.

We accomplish this using a Reed-Solomon-like encoding scheme reliant on linear algebra. We view each character as an element of the finite (Galois) field $\text{GF}(2^l)$ and thus each message as a vector in $\text{GF}(2^l)^k$. All arithmetic is performed over this finite field.

We call a matrix $F \in \text{GF}(2^l)^{n \times k}$ an *information dispersal matrix* if any $k \times k$ submatrix of F is invertible. F is *systematic* if the first k rows of F form the $k \times k$ identity matrix. Let $\vec{d} \in \text{GF}(2^l)^k$ be a message and $F \in \text{GF}(2^l)^{n \times k}$ be a systematic information dispersal matrix. Then $\vec{c} = F\vec{d}$ is a *codeword*. Since F is systematic, the first k characters of \vec{c} are the k characters of \vec{d} . The last m characters of \vec{c} are redundant or *check data*. There are a number of ways to construct a systematic information dispersal matrix. See [13] and [14] for one one example. Such a matrix may be constructed with as much redundancy as desired subject to the constraint $n \leq 2^l$, and this matrix uniquely defines the code.

3.2 Error Detection

It is easy to verify the integrity of data in this scheme. Given a length n received word, interpret the first k characters as a message. Use this message to recalculate the check data, as described above. If the computed check data matches the check data portion of the received word, then the received word is a codeword and, with high probability, the first k characters constitute the original message. However, if the check data does not match, an error has been detected.

3.3 Erasure Decoding

In some cases, a codeword may be corrupted by a simple deletion of a few characters in the word at known locations. These deletions are called *erasures* and recovery from erasures is straightforward when there are no other errors present. If characters e_1, \dots, e_f have been erased, with $f \leq m$, then we can recover the intended message in the following manner. Let \hat{c} represent the first k non-erased characters from the received word. Form \hat{F} by removing rows e_1, \dots, e_f from F ; also remove the last $m - f$ rows. \hat{F} is a $k \times k$ submatrix of F and is thus invertible. Therefore, $\vec{d} = \hat{F}^{-1}\hat{c}$.

3.4 Code Distance and Error Decoding

Recovering a message from a received word when the locations of the errors are unknown is more difficult and requires greater redundancy. Here, we will only discuss the conditions under which it is possible to recover from such *silent errors*, and leave the details of how to perform the decoding to other sources.

The *Hamming distance*, or simply distance, between two words of the same length is the number of positions at which the two words differ [4]. The *distance* d of a code is the minimum distance between any two codewords in the code. This is the minimum number of errors that must occur in order to change one valid codeword into another.

Notice from the discussion of erasure decoding in Section 3.3, that any k correct characters from a codeword are sufficient to recover the message. Therefore, any two codewords must differ from each other in at least $m + 1$ positions. For one codeword to be transformed into another, at least $m + 1$ errors must have accumulated. Thus $d = m + 1$ for this class of codes.

Without a known bias in the types of errors that accumulate in a codeword, the standard approach is to decode a received word to the codeword that is nearest in Hamming distance. If a codeword has accumulated t errors during storage or transmission, then the received word is a distance t from the codeword. If $t < \frac{d}{2}$ then no other codeword is closer to the received word than the original codeword. The received word will be correctly decoded to the original codeword. On the other hand, if $t \geq \frac{d}{2}$ then some other codeword may be closer to or as close to the received word as the original codeword. If so, the decoding will be incorrect. For our codes, if $t < \frac{d}{2} = \frac{m+1}{2}$ silent errors have accumulated in a given received word, then we can always correctly decode to the original codeword. $t < \frac{m+1}{2}$ is the *error bound* for this code.

For t small, e.g. $t = 1$ or $t = 2$, it is reasonably efficient to simply check whether each word that is distance t from the received word is a codeword. If one is, decode to that codeword. For larger t , a more sophisticated algorithm is required to decode efficiently. The Welch-Berlekamp algorithm is suggested [17, 2].

3.5 Simultaneous Erasure and Silent Error Decoding

These codes can also correct simultaneous erasures and silent errors, for m sufficiently large. If the received word contains f erasures and t silent errors then it may always be decoded correctly for $2t + f < m + 1$. Exhaustive search of nearby words is recommended for t small, and the Welch-Berlekamp algorithm for t large.

3.6 ECC for Data Storage

A primary use of ECC in computer systems is to protect stored data from corruption or device failure. RAID systems are often used for this task. A RAID (Redundant Array of Independent Disks) is a collection of total $T = M + N$ storage devices with the capacity to hold N devices worth of data. As with general ECC, the stored data is broken into l bit characters, and the l bit characters are

grouped into length $k = N$ messages. Each character in a message is stored on a different one of the N data devices. $m = M$ characters of check data are generated per Section 3.1; these characters are stored across the remaining M check devices. Together with the Reed-Solomon coding scheme, we refer to this setup as RS(T, M) by analogy with the simpler schemes such as RAID0–6.

The failure of a device in the RAID corresponds to an erasure. So long as no more than M devices fail simultaneously, full data recovery is possible using the technique outlined in Section 3.3. Undetected sector failures and erroneous device read or write operations might contribute silent errors to a codeword. These may also be detected and corrected per Sections 3.2, 3.4, and 3.5.

4 The Computational Cost of ECC

ECC is not free but comes with certain computational costs. In this section we will analyze these costs for various ECC-related tasks.

In a computer system, we usually must handle a large amount of data all at once, particularly when preparing data for RAID storage. Let $D \in GF(2^l)^{k \times p}$ be a matrix of such data. Each column of D corresponds to a message \vec{d} ; p is the number of messages to be handled; and the i th row of D consists of all characters in the i th message position in all of the messages. $C = FD$ is the matrix of all codewords, including both message and check data. The first k rows of C are identical to D , and the last m rows of C are check data.

4.1 Generating Check Data

The check characters in the codewords C are computed by multiplying D by the last m rows of F . This requires mkp multiplications and $m(k-1)p$ additions over $GF(2^l)$. This is essentially m multiplications and additions for each character of message data.

4.2 Recovery from Erasures

Recovery from data erasures will be of particular interest in the application of ECC to storage systems, as discussed in Section 3.6. In this case, a failed storage device corresponds to the erasure of a row of data from the matrix C . We now give a detailed account of the computation needed to recover from such erasures.

Notice that since F is systematic, it has the form

$$F = \begin{bmatrix} I_k \\ \star \end{bmatrix},$$

where I_k is the $k \times k$ identity matrix, and \star some matrix. Assume there have been f erasures of rows of data from

the first k rows of C corresponding to message data, and thus $s = k - f$ message characters survive in each codeword. So long as no more than $m - f$ of the check characters are erased, we can recover the erased message data. We need only k characters total from the codewords to recover this data, so we may discard extra check characters if fewer than m total characters were erased.

Construct \hat{C} as k characters (rows) of C that were not erased, including all available message characters. If Q is obtained from the $n \times n$ identity matrix by removing the m rows corresponding to erased (or ignored) characters, then $\hat{C} = QC$. Construct \hat{D} as a permutation of D so that good data drives appear first in D and erased or ignored data appears last. We may denote this by $\hat{D} = PD = \begin{bmatrix} X \\ Y \end{bmatrix}$ where P is the appropriate permutation matrix, X is the good data and Y is the lost data. If we then set $\hat{F} = QFP^T$ (noting that $P^T = P^{-1}$) we have the reduced system $\hat{F}\hat{D} = \hat{C}$. \hat{F} has the additional block structure

$$\hat{F} = \begin{bmatrix} I_k & 0 \\ A & B \end{bmatrix},$$

so we have the system of equations

$$\begin{bmatrix} I_k & 0 \\ A & B \end{bmatrix} \begin{bmatrix} X \\ Y \end{bmatrix} = \begin{bmatrix} X \\ W \end{bmatrix}$$

where W is the matrix of remaining check data. This can be rewritten as $AX + BY = W$. Y is the unknown lost data, and all other variables are known, so we simply solve this system for Y : $Y = B^{-1}(W - AX)$.

It is the cost of computing Y via this formula in which we are interested. We will assume that the operation cost associated with constructing B , W , A , and X are negligible, as through clever referencing of F and C they need not be explicitly constructed. There are three steps in computing Y with operations counts as follows: (1) Computation of AX requires sf Galois Field multiplies and $(s - 1)fp$ GF additions. (2) Subsequent computation of $W - AX$ requires fp additions. (3) Solving $BY = W - AX$ for Y via LU factorization of B may be further broken into three steps:

1. Computing the LU factorization of B without pivoting requires $\frac{1}{3}(f^3 - f)$ additions, $\frac{1}{3}(f^3 - f)$ multiplies, and $\frac{1}{2}f(f - 1)$ divides [15].
2. Solving $LZ = W - AX$ for Z using forward substitution requires $(f - 1)p + \frac{1}{2}(f^2 - 3f + 1)p$ additions, and $\frac{1}{2}(f^2 - 3f + 1)p$ multiplies.
3. Solving $UY = Z$ for Y using backward substitution requires $(f - 1)p + \frac{1}{2}(f^2 - 3f + 1)p$ additions, $\frac{1}{2}(f^2 - fm + 1)p$ multiplications, and fp divisions.

The overall operation count is:

- Additions (including subtractions): $fp(f + s - 1) - l + \frac{1}{3}(f^3 - f)$;
- Multiplications: $fp(f + s - 3) + l + \frac{1}{3}(f^3 - f)$;
- Divides: $fp + \frac{1}{2}f(f - 1)$.

RAID systems are generally measured by their throughput, that is, how many bytes (characters) can be accepted or delivered within a fixed time interval. With this in mind, the most useful perspective on the cost of computation is with regard to each data byte that is accepted or delivered, *i.e.* the number of original data bytes in the system which is the number of data bytes in D , Nl . We can think of the operation counts listed above as a cost per original byte plus some overhead associated with computing the LU factorization of B . Table 3 summarizes these costs, also using the relationship $k = f + s$ to further simplify expressions.

Count	
+, -	$fkp - fp - p + \frac{1}{3}(f^3 - f)$
·	$fkl - 3fp + p + \frac{1}{3}(f^3 - f)$
÷	$fp + \frac{1}{2}f(f - 1)$
Count per byte of original data	
+, -	$f - \frac{f+1}{k}$
·	$f - \frac{3f-1}{k}$
÷	$\frac{f}{N}$
Overhead	
+, -	$\frac{1}{3}(f^3 - f)$
·	$\frac{1}{3}(f^3 - f)$
÷	$\frac{1}{2}f(f - 1)$

Table 3: Summary of the count of operations required to recover lost data.

In summary, if f data devices are lost, then it costs about f additions and multiplies and < 1 divide per byte of original data requested to read data plus an additional f^3 additions and multiplies and f^2 divides of overhead costs associated with computing the LU factorization. When $f \ll p$ as is typically the case, the cost of f additions and multiplies per original data byte is the dominating cost of data delivery. As discussed in Section 4.1, it costs about m additions and multiplies per original data byte to generate (or regenerate) the check drives. Thus the cost of preparation for the failure of m drives is essentially the same as the cost of recovering from the failure of m drives: m additions and m multiplications.

4.3 System performance and GalOPS

Computational devices such as the CPU are capable of performing a certain number of Galois field operations per second. This determines the speed of Reed-Solomon style ECC. In a general purpose CPU the circuitry that performs integer arithmetic (modulo a power of 2, typically 2^k , where $k = 8, 16, 32$ and 64) is used to simulate the Galois field operations. $GF(2^l)$ for small l is a finite set of cardinality small enough for a lookup table approach to multiplication and division. Additions may be directly performed by the existing integer arithmetic logic. Today's CPU's lack a dedicated hardware implementation of Galois field operations, but this situation may change. To make reasonable choices of the parameters of the system, ECC system designers face the problem of estimating how much computer resources the main algorithm would consume. This includes the silicon and CPU cycles. The absolute operation counts presented in this section can be a starting point of evaluating system performance. However, due to pipelining and branch prediction, more than 1 Galois field operation will be performed per clock cycle. In fact, the number can easily be as high as 16–32 on a real computer system, as evidenced by the data presented in Section 2.

We propose rating computer systems according to the number of Galois field operations per second that the system can perform. Necessarily, the measurement would need to be performed using some standardized benchmarking software.

In the past, similar synthetic measures have been devised for floating-point arithmetic, and the most popular measure in the High-Performance Community is the number of FLOPS, Floating point Operations per Second, that a processor can perform. For a 2.4GHz processor, it is not uncommon to achieve 25 GigaFLOPS, an order of magnitude difference between the clock speed and the FLOPS count, on standard Linear Algebra benchmarks (LINPACK).

A similar measure for the I/O subsystem only is called IOPS and is used in the IOMeter software originally from the Intel Corporation [8].

We propose a similar measure expressed in GalOPS, Galois Operations per Second (pronounced *G-ah-lops*), to evaluate the performance of a computer system from the point of view of its capability to perform Galois field arithmetic. Since the essence of the erasure coding algorithm we presented is linear algebra over the finite field $GL(2^l)$ or in the vector space $GL(2^l)^k$, we could use the Reed-Solomon code implementation introduced in Section 2 as one such measure. Of course, the measure would depend on parameters, such as l and k .

In the era of virtual design, GalOP ratings of computer systems could not only be obtained for existing, but also

future hardware (using existing simulators), and help in controlling costs before large systems supporting ECC, such as cloud infrastructures, are even built or deployed.

5 Design and Reliability of RAID

The engineering design of RAID systems is about increasing *reliability* by using *redundancy*, and yet it involves compromises, such as keeping the design cost down by using the smallest number of devices which satisfying the reliability objectives. We must have quantitative measures of reliability and, short of conducting massive, costly experiments, we must rely upon mathematical models to predict, for instance, what fraction of deployed RAID systems would fail in, say, 5 years.

A simple quantitative model of reliability of a $RS(T, M)$ system should represent reliability as function of its fundamental parameters: the number of data drives N , the number of check drives M , the total number of drives $T = M + N$, and the reliability of a single device, represented by the *failure rate* λ , which is assumed constant. Modeling various scenarios leading to RAID failure is a subject of a significant number of publications [10, 11, 3, 6]. A popular reliability measure is the *mean time to data loss* (MTTDL). In the case of an isolated RAID system which can only fail due to random loss of devices happening with rate λ and which is never repaired, an exact formula for MTTDL exists [6]:

$$\text{MTTDL} = \frac{1}{\lambda} \sum_{k=0}^M \frac{1}{1-k/T} \frac{1}{T} \approx \frac{1}{\lambda} \int_0^{M/T} \frac{1}{1-u} du.$$

It is not hard to see that for large T the following approximation can be made:

$$\text{MTTDL} \approx \lambda \log \frac{1}{R}. \quad (1)$$

where $R = \frac{N}{T}$ is the *rate*, the fraction of data devices in the RAID. Solving for R we obtain:

$$R = e^{-\frac{\text{MTTDL}}{\lambda}} \quad (2)$$

This yields a simple, prescriptive rule which tells us what portion of the devices in our system should be data devices vs. check devices to achieve a desired MTTDL by determining R via the formula above. λ , the failure rate of an individual device is typically provided by the device manufacturer. Notice that λ is the inverse of the MTTDL for a single device. RAID failure modeling literature provides more rules of this sort, taking into account common sources of failure, such as sector errors, maintenance schedules and human error [6].

5.1 Benefits of $M > 2$

Note that 2 check devices are used in the familiar RAID 6 configuration. The main objection to systems with this few check devices is their vulnerability to not only device failure but also silent data corruption. We can easily evaluate the problem remembering the inequality $2t + f < M + 1$ from Section 3.5, which must hold in order to recover from errors. Thus, RAID with $M = 1$ cannot recover from just a single silent data error (unnoticed corruption of 1 byte of $M + N$ of the codeword). When $M = 2$, 1 failed device and 1 silent data error cause the system to lose data. Since it is expected that every RAID6 will have at least one failure in its lifetime, RAID6 is not resilient to silent errors. In fact, a RAID6 that encounters a silent error during reconstruction will silently corrupt additional data. To be resilient in the face of silent errors, which have been carefully documented in large systems [9], a minimum of $M=3$ check drives are required.

5.2 Large M solves the Big Data problem

The main benefits of a design that supports a higher check device count are lowered cost and increased reliability for large (Big Data) systems. By matching M with N we may build optimal storage systems for arbitrarily large data and arbitrarily strong reliability requirements, and thus solve the Big Data problem.

For example, no modern RAID vendor supports a RAID 6 configuration of 50 drives or more. They recognize that the likelihood of data loss is too high. Instead, they deploy less efficient, more costly configurations such as 10+2 (i.e. $M = 10$ and $N = 2$), replicated 5 times.

Using more check drives allows the deployment of larger configurations, for example, 50+5, that require fewer total components and need less frequent service. The service requirement can be delayed until the risk of data loss warrants a service call.

Larger systems with more check drives lower the acquisition, operation and service costs, while simultaneously increasing the protection from data loss.

6 Conclusions

The world is filled with sophisticated CPU devices. These range from those in our cell phones and digital cameras to our automobiles, airplanes, communication processors and business equipment. Today, it is normal for data to remain unprotected as it travels around these systems, and as such, is lost. Who has not seen “high-definition” video disintegrating to a collection of large, colorful squares²,

²The proper term is “pixelation”. There are other causes of pixelation besides data loss. Those cannot be addressed with ECC because they are artifacts of “normal” lossy codec operation, which gives priority to

or heard audio dropouts, or experienced corrupted files? This state of affairs is no longer necessary to tolerate. Data need not be lost again³. With the addition of a simple software ECC algorithm, data can be protected from its “birth”, whether it originates in a camera, or as the result of a business calculation, or as part of the research that will affect our future health and security.

We call our variant of this solution *Virtual ECC*. We identified how, utilizing the ubiquity of CPU’s around us, to eliminate many sources of noise, both identified and silent, and provided hard measurements of real processors computing very reliable codewords. We described a practical implementation as a kernel module in the forthcoming Windows 8 platform. We presented benchmarks showing that the approach is not only viable, but it provides excellent performance and reliability characteristics, exceeding those of typical hardware ECC.

References

- [1] ATTO Technology, Inc. Disk Benchmark—ATTO. http://www.attotech.com/products/product.php?sku=Disk_Benchmark, 2012. Software.
- [2] Peter Gemell and Madhu Sudan. Highly resilient correctors for polynomials. *Information Processing Letters*, 43(4):169–174, September 1992.
- [3] Kevin M. Greenan, James S. Plank, and Jay J. Wylie. Mean time to meaningless: MTDDL, Markov models, and storage system reliability. In *HotStorage '10: 2nd Workshop on Hot Topics in Storage and File Systems*, Boston, June 2010.
- [4] R. W. Hamming. Error detecting and error correcting codes. *Bell System Technical Journal*, 29:147–160, April 1950.
- [5] F. J. MacWilliams and N. J. A. Sloane. *The Theory of Error-Correcting Codes, Part I*. North-Holland Publishing Company, Amsterdam, New York, Oxford, 1977.
- [6] Sarah Edge Mann, Michael Anderson, and Marek Rychlik. On the Reliability of RAID Systems: An Argument for More Check Drives. *arXiv.org*, cs.PF, February 2012.
- [7] Microsoft. <http://code.msdn.microsoft.com/windowshardware/WDKStorPortVirtualMiniport-973650f6>.

maintaining fixed bitrate over image quality.

³At least on time scales so long that an average person will *never* see important data loss during their lifetime.

- [8] Intel Corporation (original version of 1998). Iometer. <http://www.iometer.org/>, 1998–2012. Software.
- [9] Bernd Panzer-Steindel. Data integrity. Technical report, CERN/IT, April 2007.
- [10] Jehan-François Paris, Ahmed Amer, Darrell D. E. Long, and Thomas J. E. Schwarz. Evaluating the Impact of Irrecoverable Read Errors on Disk Array Reliability. In *Proceedings of the IEEE 15th Pacific Rim International Symposium on Dependable Computing (PRDC09)*, November 2009.
- [11] Jehan-François Paris, Thomas J. E. Schwarz, Darrel D. E. Long, and Ahmed Amer. When MTDLs Are Not Good Enough: Providing Better Estimates of Disk Array Reliability. In *Proceedings of the 7th International Information and Telecommunication Technologies Symposium (I2TS '08)*, December 2008.
- [12] W. W. Peterson and E. J. Weldon, Jr. *Error-Correcting Codes*. The MIT Press, Cambridge, Massachusetts, second edition, 1972.
- [13] James S. Plank. A Tutorial on Reed-Solomon Coding for Fault-Tolerance in RAID-like Systems. *Software – Practice & Experience*, 27(9):995–1012, September 1997.
- [14] James S. Plank and Y. Ding. Note: Correction to the 1997 Tutorial on Reed-Solomon Coding. *Software – Practice & Experience*, 35(2):189–194, February 2005.
- [15] Lloyd N. Trefethen and David Bau. *Numerical Linear Algebra*. SIAM: Society for Industrial and Applied Mathematics, June 1997.
- [16] J. H. van Lint. *Introduction to Coding Theory*. Springer-Verlag, New York, 1982.
- [17] Lloyd R. Welch and Elwyn R Berlekamp. Error Correction for Algebraic Block Codes, December 1986. United States Patent. Patent number: 4,633,470.

EXHIBIT J

SwiftStack Blog



[Home](#) / [SwiftStack Blog](#) /

Save Space: the final frontier - Erasure Codes with OpenStack Swift

Save Space: The Final Frontier - Erasure Codes With OpenStack Swift

Jul 10th,
2013



Today we're really excited to announce an initiative to introduce erasure codes in OpenStack Swift. Swift currently uses replicas, but a question has come up – could we save space by using erasure codes?

This initiative enables deployers to store data with erasure coding instead of or in addition to Swift's 3-replica model. Though using 3 replicas provides for excellent performance and availability, it's incurred in both the acquisition and operating cost of storage

hardware. Swift has already enabled many companies to radically lower their storage costs with commodity hardware and the introduction of erasure coding within Swift will enable costs to drop even further.

The development of this feature will proceed with the same open mindset that has guided the OpenStack project from its inception. Just like all projects within OpenStack, Swift has many contributors. The companies who are heavily involved with Swift include SwiftStack, Rackspace, Red Hat, IBM and HP.

For erasure coding, multiple companies — Intel, SwiftStack, Box and EVault are committing effort for this specific project –

“Intel is excited to support the development of an erasure code solution for OpenStack Swift with the Swift development community. Helping our customers reduce the size of data on disk by up to half versus regular triple replication, helps decrease their costs by more than 50%. Erasure code solutions reduce both hardware requirement costs as well as the power and cooling required to run that hardware,” says Bev Crair, Intel Storage Division GM. “Erasure code is a technology that is long overdue and Intel is pleased to be supporting efforts to promote and use it in cloud environments like OpenStack Swift.”

“EVault is excited to work with Swiftstack and the broader OpenStack Object Storage community to add erasure codes.” says George Hoenig, Vice President, Products & Services at EVault. “Erasure codes, particularly for write intensive workloads, will enable users to deploy systems using less storage and bandwidth than replicated systems of similar durability.”

Starting from a production-grade system

By using Swift as a starting point, we stand on the shoulders of the existing, battle-hardened mechanisms that Swift already has.

We are also enlisting some of the thought leaders in information theory and erasure coding who are contributing code and guidance for this project.

The design goal is to be able to have erasure-coded storage plus replicas coexisting in a single Swift cluster. This will allow a choice in how to store data and will allow applications to make the right tradeoffs based on their use case.

There are already proposals and code on the table for this effort. And we will be collaborating over these designs over the coming months to build a solution to best meet the needs of the Swift deployers.

Development as a community

We have a big project ahead of us. But we have rallied as a community before and have pulled off some big efforts. For example, [region support](#) is now included in the [latest version](#) of Swift which allows a cluster to span distant data centers.

This effort continues to demonstrate the focus of the Swift project – to grow an already great object storage system into the new areas where haven't gone before. With continued efforts such as this, Swift is well on its way ensure your data can “live long and prosper”.



Joe Arnold
CEO, SwiftStack

[@joearnold](#)
joe@swiftstack.com

Categories

[OpenStack](#), [OpenStack Swift](#), [PlanetOpenstack](#), [SwiftStack](#)

[Global Clusters and more:
Swift 1.9.0](#)

[Erasure Codes with
OpenStack Swift –
Digging Deeper](#)

Favorite Posts

[Kinetic Motion with Seagate and OpenStack Swift](#)

[A Closer Look at Software-Defined Storage](#)

[Gartner on OpenStack Swift](#)

[The Top 3 New Swift Features in OpenStack Folsom](#)

[A Globally Distributed OpenStack Swift Cluster](#)

[Video: How OpenStack Swift Handles Hardware Failures](#)

[SwiftStack welcomes John Dickinson](#)

[Swift Capacity Management](#)

CloudStack going Apache 2

Recent Posts

In 10 Years, all Storage Will Be Object Storage! There, I said it.

Join us at the Cloud Computing Expo in New York

Everything is Object Storage Roundup: Gartner's IaaS Magic Quadrant, IBM Says Goodbye to NetApp, and More Proof Object Storage Is In

OpenStack Swift June Hackathon in Denver

Q&A with Joe Arnold on the new O'Reilly published book, "Object Storage with Swift"

Archived Posts

Blog Archives

Follow @SwiftStack



Tweet 25

+1 7

Share 8

Like Share 12 people like this. Be the first of your friends.

Comments

6 Comments

SwiftStack

Login ▾

Sort by Best ▾

Share Favorite

LATEST BLOG POSTS

Jun 17
[In 10 Years, all Storage Will Be Object Storage!](#)
[There, I said it.](#)

Jun 11
[Join us at the Cloud Computing Expo in New York](#)

Jun 09
[Everything is Object Storage](#)

SwiftStack Architecture

[Roundup: Gartner's IaaS Magic Quadrant, IBM Says Goodbye to NetApp, and More Proof Object Storage Is In](#)

© 2014 SwiftStack Inc.

San Francisco, CA

contact@swiftstack.com

EXHIBIT K



RELEASED FOR PUBLICATION

StreamScale Provides Notice of Ownership of Fastest Erasure Code Technology Disclosed at FAST'13

*Technology Wrongly Identified as "Open Source:"
Technology Only Available for Use Through Properly Executed Licensing
Agreements With StreamScale*

Los Angeles, CA – July 23, 2013: [StreamScale](#), a leading developer providing technology to protect storage systems from data loss and corruption, announced today it has discovered publication of alleged "open source" materials that include the Company's confidential and patent-pending technology. This StreamScale technology is not "open source" and was improperly disclosed in publications and presentations at FAST'13, the 11th annual USENIX conference on file and storage technologies.

In particular, StreamScale has learned that one of its consultants was an author of various papers including one entitled "Screaming Fast Galois Field Arithmetic Using Intel SIMD Instructions" publicly presented at FAST'13. These papers contain technology StreamScale shared with that consultant under a confidential relationship. The technology is protected intellectual property owned by StreamScale, and it is not "open source." StreamScale announced it will enforce its rights to this technology and parties may not use this technology in any way without a proper license from StreamScale.

"We have recently become aware that StreamScale confidential information and trade secrets were disclosed in papers delivered at FAST'13 and elsewhere without StreamScale's approval or knowledge," said Michael H. Anderson, president and CEO, StreamScale. "A paid consultant revealed information provided to him by StreamScale in violation of the terms of his contract. It is clear that the research and resultant findings disclosed by the consultant and his co-authors were a direct result of StreamScale's proprietary IP. StreamScale's technologies are not only the subject of trade secret protection but are also subject to copyright protection and at least two pending patent applications."

StreamScale patent applications can be found at the US Patent & Trademark Office website and [the company's website](#):

- 1) ACCELERATED ERASURE CODING SYSTEM AND METHOD (U.S. Patent Application No. 20130173996)
- 2) USING PARITY DATA FOR CONCURRENT DATA AUTHENTICATION, CORRECTION, COMPRESSION, AND ENCRYPTION (U.S. Patent Application No. 20130173956)

Continued Anderson, "Do not be misled by claims that the technology is "open source" and do not assume that such information downloaded from USENIX or University websites is unprotected. We have asked these organizations to remove

the content from their websites and conference proceedings. I am informing any company or person that uses this protected technology without license from StreamScale does so at their own risk.”

In addition to the paper presented at FAST’13, StreamScale has identified a number of other papers from this consultant which include the patent pending technology and must not be used without license from StreamScale. While it is unknown how many papers have included the information, here are several that people should understand include StreamScale’s protected information:

- M. Blaum and J. S. Plank, Construction of two SD Codes, arXiv: 1305.1221, May, 2013.
- J. S. Plank and M. Blaum, “Sector-Disk (SD) Erasure Codes for Mixed Failure Modes in RAID Systems,” Technical Report CS-13-708, University of Tennessee EECS Department, May, 2013.
- J. S. Plank, “Open Source Encoder and Decoder for SD Erasure Codes - Revision 2.0,” Technical Report CS-13-707, University of Tennessee EECS Department, May, 2013.
- J. S. Plank, “Open Source Encoder and Decoder for SD Erasure Codes,” Technical Report CS-13-704, University of Tennessee EECS Department, January, 2013.
- J. S. Plank, E. L. Miller and W. B. Houston, “GF-Complete: A Comprehensive Open Source Library for Galois Field Arithmetic, Revision 0.1,” Technical Report CS-13-703, University of Tennessee EECS Department, January, 2013.

At the core of the StreamScale technology is high performance erasure coding that optimizes Galois Field (GF) arithmetic. StreamScale discovered how to utilize specific instructions in the Intel SSE3 SIMD instruction set to perform GF arithmetic so that it is only limited by the L2/L3 cache, providing up to a 12x improvement over the best performing existing XOR codes. As reflected in patent filings dating back to 2011, implementing erasure coding with StreamScale technology enables the creation of the best performing most reliable storage systems that eliminate data loss and corruption due to disk failure, service errors, silent data corruption, and unrecoverable read errors while increasing system performance.

“Unfortunately we have read comments online that individuals and companies are using these protected technologies based on these papers to optimize storage systems and other products without license from StreamScale,” concluded Anderson. “We have the utmost respect for USENIX and technology companies in the industry and anticipate they will do the right thing by removing the improperly disclosed information from their websites and conference proceedings and seek a license agreement from us or remove the StreamScale IP for their solutions.”

About StreamScale

[StreamScale](#) leads the industry in providing technology to protect storage systems from data loss and corruption. The [Big Parity® Verified Erasure Coding® system](#) is available to all data storage manufacturers and system integrators. By including Big

Parity in their storage system manufactures can achieve 10 orders of magnitude better RAID reliability and up to 30X faster system performance.

Press Contacts

Curtis Chan
COGNITIVE IMPACT
Office: +1 714.447.4993
Fax: +1 714.447.6020
E-mail: curtis@cognitiveimpact.com

EXHIBIT L

StreamScale

O'Shea, Michael A. [moshea at hunton.com](mailto:moshea@hunton.com)

Wed Apr 29 15:15:09 UTC 2015

- Previous message: [MRE - neutron-*aas](#)
- Messages sorted by: [\[date \]](#) [\[thread \]](#) [\[subject \]](#) [\[author \]](#)

To: [technical-board at lists.ubuntu.com](mailto:technical-board@lists.ubuntu.com)<[mailto:technical-board at lists.ubuntu.com](mailto:technical-board@lists.ubuntu.com)>

I am writing to you today regarding Ubuntu's recent republication and redistribution of the libraries JErasure 2.0 and GF_Complete, authored by Dr. James Plank.

These libraries were the subject of confidential trade secret litigation between my client, StreamScale Inc., and the author, Dr. James Plank (JAMS case #1220047807). Subsequent to that litigation, Dr. Plank removed JErasure 2.0 and GF_Complete from publication, leaving only an archived copy for non-commercial use, and publicly announced that he would no longer support either library (<http://web.eecs.utk.edu/~plank/plank/www/software.html><https://urldefense.proofpoint.com/v2/url?u=http-3A-web.eecs.utk.edu-7Eplank_plank_www_software.html&d=AwMPag&c=jxhwBfk-KSV6FFIot0PGng&r=OsbfxqvwkClXeDQMkz1i9dGxH4gYoBogKZbCGO22AwM&m=nwTBnMf0m1nkQVj42tenme0z2BKWCLL0XbbEdhy3Zc&s=aclLu9_Bo7RA0XpRp7M_xyVV3894VL>)

We believe that even a cursory review of the facts surrounding these libraries will make it clear to you that they do not constitute "Free Software" by any reasonable definition. StreamScale developed an extensive litigation database related to the libraries JErasure 2.0 and GF_Complete, including patent claim charts, trade secret listings, copyright infringement analysis, damage estimates and citations of relevant case law. We could make these materials available to you under a suitable NDA at your request to provide any reasonable clarification you might need.

We respectfully ask that you voluntarily remove and not republish the libraries JErasure 2.0 and GF_Complete, as well as any other packages or releases that incorporate them or depend on them. We believe these packages and releases include: libjerasure-dev, libjerasure2, libgf-complete-dev, libgf-completel, liberasurecode-dev, liberasurecode1, Pyeclib, CEPH (see ceph/src/erasure-code/jerasure) and Swift 2.3.0.

Thank you for your consideration in this matter, and please feel free to contact me regarding any questions you might have.

Best,

Michael O'Shea

Bio<<http://webdownload.hunton.com/esignature/bio.aspx?U=12011>> vCard<<http://webdownload.hunton.com/esignature/vcard.aspx?U=12011>>

[Hunton and Williams]

Michael O'Shea

Partner
[moshea at hunton.com](mailto:moshea@hunton.com)<[mailto:moshea at hunton.com](mailto:moshea@hunton.com)>

Hunton & Williams LLP
2200 Pennsylvania Avenue, NW
Washington, DC 20037
Direct: 202.419.2183
Fax: 202.778.7434
www.hunton.com<<http://www.hunton.com/>>

----- next part -----
An HTML attachment was scrubbed...
URL: <<https://lists.ubuntu.com/archives/technical-board/attachments/20150429/1940b14d/attachment-0001.html>>
----- next part -----
A non-text attachment was scrubbed...
Name: image001.jpg
Type: image/jpeg
Size: 5414 bytes
Desc: image001.jpg
URL: <<https://lists.ubuntu.com/archives/technical-board/attachments/20150429/1940b14d/attachment-0001.jpg>>

-
- Previous message: [MRE - neutron-*aas](#)
 - Messages sorted by: [\[date \]](#) [\[thread \]](#) [\[subject \]](#) [\[author \]](#)

[More information about the technical-board mailing list](#)

EXHIBIT M

StreamScale, Inc.



July 7, 2021

Via E-mail

Benjamin R. Ostapuk
Intel Corporation
2200 Mission College Blvd.
Santa Clara, CA 95054-1537
benjamin.r.ostapuk@intel.com

Re: Notice of Infringement of StreamScale Patents

Dear Mr. Ostapuk:

I write on behalf of StreamScale, Inc. (“StreamScale”) to notify Intel Corporation (“Intel”) that it is inducing, for example, Cloudera, Inc., ADP, Inc., Experian plc, and Wargaming (Austin), Inc. (collectively, “the Direct Infringers”) to infringe certain StreamScale United States Patents.

Intel is inducing infringement of: (1) U.S. Patent No. 8,683,296; (2) U.S. Patent No. 9,160,374; (3) U.S. Patent No. 9,385,759; (4) U.S. Patent No. 10,003,358; (5) U.S. Patent No. 10,291,259; and (6) U.S. Patent No. 10,666,296 (collectively, “the StreamScale Patents”). The StreamScale Patents are continuations of one another and each of the StreamScale Patents relates to the field of accelerated erasure coding.

Intel actively markets and instructs the Direct Infringers to create erasure coding systems using Intel’s Intelligent Storage Acceleration Library (“ISA-L”). For example, Intel (i) maintains a website to promote ISA-L, including to the Direct Infringers,¹ (ii) produces videos regarding ISA-L and its use that are available to the Direct Infringers on the Intel website,² (iii) describes case studies on big data optimization using ISA-L that are available to the Direct Infringers on the Intel website, (iv) hosts articles, blog posts, and webinars regarding the use of ISA-L that are available to the Direct Infringers on the Intel website, and (v) publishes and makes available an API Reference Manual for ISA-L³ that is available to the Direct Infringers, which it updates regularly.⁴ Intel further offers the Direct Infringers technical support for ISA-L.

Intel designed ISA-L to be used with other components that, when combined with hardware, practice one or more claims of each of the StreamScale Patents. ISA-L is a collection of functions used in storage applications, including functions pertaining to erasure codes that implement a general Reed-Solomon type encoding for blocks of data to protect against erasure of whole blocks.⁵ The claims of the StreamScale Patents require, variously, data and check matrices to hold original and check data in memory, respectively. These matrices correspond with parameters described in ISA-L documentation associated with, for example, the function `ec_encode_data`.⁶ The “data” parameter corresponds to the data matrix in the claims of the StreamScale Patents. The “coding” parameter corresponds to the check matrix in the claims of the

¹ *E.g.*, Intel, Intel® Intelligent Storage Acceleration Library, available at <https://software.intel.com/content/www/us/en/develop/tools/isa-l.html> (last visited May 24, 2021).

² *See, e.g.*, Intel, Erasure Code and Intel® Intelligent Storage Acceleration Library (Intel® ISA-L, available at <https://www.intel.com/content/www/us/en/products/docs/storage/erasure-code-isa-lsolution-video.html>) (last visited May 24, 2021).

³ *See, e.g.*, Intel, Intel® Intelligent Storage Acceleration Library (Intel® ISA-L) Open Source Version, API Reference Manual (ver. 2.8, Sept. 27, 2013), available at https://01.org/sites/default/files/documentation/isa-l_open_src_2.8_0.pdf (last visited May 24, 2021).

⁴ *See, e.g.*, Intel, Intel® Intelligent Storage Acceleration Library (Intel® ISA-L), API Reference Manual (ver. 2.23.0, June 29, 2018), available at https://01.org/sites/default/files/documentation/isa-l_api_2.23.0.pdf (last visited May 24, 2021).

⁵ *E.g.*, Intel Corporation, Intel® Intelligent Storage Acceleration Library (Intel® ISA L) Open Source Version, API Reference Manual §§ 1.2-1.3 (Version 2.14, July 16, 2015).

⁶ *E.g., id.* § 5.1.2.1.



StreamScale Patents. And the “gftbls” parameter corresponds to the encoding matrix referenced in the claims of the StreamScale Patents. Similarly, the claims of the StreamScale Patents variously require a parallel multiplier, which ISA-L provides in various permutations of **vect_dot_prod** functions.⁷

Given Intel’s enormous size, sophistication, and resources, StreamScale is confident Intel can deeply appreciate how ISA-L indirectly infringes the claims of the StreamScale Patents. StreamScale requests that Intel cease its above-identified infringing activities relating to the StreamScale Patents. In the alternative, StreamScale is willing to discuss an appropriate license to StreamScale’s inventions and how StreamScale and Intel can work together to move the industry forward with new technology and innovations.

Best regards,

Bryan D. Richardson
Chief Intellectual Property Officer

cc: Sonal Mehta, Esq. (sonal.mehta@wilmerhale.com)

⁷ *E.g.*, Intel, ISA-L, `ec_highlevel_func.c`, available at https://github.com/intel/isa-l/blob/master/erasure_code/ec_highlevel_func.c (last visited Nov. 23, 2020) (lines 33–68).