Erasure Codes for Storage Systems

A Brief Primer

JAMES S. PLANK

James S. Plank received his BS from Yale University in 1988 and his PhD from Princeton University in 1993. He is a professor in the Electrical Engineering and Computer Science Department at the University of Tennessee, where he has been since 1993. His research interests are in fault-tolerance, erasure coding, storage systems, and distributed computing. He is a past Associate Editor of IEEE Transactions on Parallel and Distributed Computing, and a member of the IEEE Computer Society. plank@cs.utk.edu

Storage systems have grown to the point where failures are inevitable, and those who design systems must plan ahead so that precious data is not lost when failures occur. The core technology for protecting data from failures is erasure coding, which has a rich 50+ year history stemming communication systems, and as such, can be confusing to the storage systems community. In this article, I present a primer on erasure coding as it applies to storage systems, and I summarize some recent research on erasure coding.

Storage systems come in all shapes and sizes, but one thing that they all have in common is that components fail, and when a component fails, the storage system is doing the one thing it is not supposed to do: losing data. Failures are varied, from disk sectors becoming silently corrupted, to entire disks or storage sites becoming unusable. The storage components themselves are protected from certain types of failures. For example, disk sectors are embedded with extra-correcting information so that a few flipped bits may be tolerated; however, when too many bits are flipped, or when physical components fail, the storage system sees this as an erasure: the storage is gone!

To deal with these failures, storage systems rely on erasure codes. An erasure code adds redundancy to the system to tolerate failures. The simplest of these is replication, such as RAID-1, where each byte of data is stored on two disks. In that way any failure scenario may be tolerated, so long as every piece of data has one surviving copy. Replication is conceptually simple; however, it consumes quite a lot of resources. In particular, the storage costs are doubled, and there are scenarios in which two failed storage components (those holding both copies of a piece of data) lead to data loss.

More complex erasure codes, such as the well-known Reed-Solomon codes, tolerate broader classes of failure scenarios with less extra storage. As such, they are applicable to today's storage systems, providing higher levels of fault-tolerance with less cost. Unfortunately, the field of erasure coding traces its lineage to error correcting codes (ECC) in communication systems, where they are used to solve a similar-sounding but in reality quite different problem. In communications, errors arise when bits are corrupted silently in a message. This differs from an erasure, because the location of the corruption is unknown. The fact that erasures expose the location of the failure allows for erasure codes to be more powerful than ECCs; however, classic treatments of erasure codes present them as special cases of ECCs, and their application to storage systems is hard to glean.

In this article, I explain erasure codes in general as they apply to storage systems. I will first present nomenclature and general erasure coding mechanics, and then outline some common erasure codes. I then detail some of the more recent research results concerning erasure codes and storage systems. I provide an annotated bibliography at the end of this article so that the interested reader may explore further.

The Mechanics of Simple Codes

Let's assume that our storage system is composed of n disks. We partition them into k disks that hold user data so that m=n−k disks hold coding information. I refer to them as data and

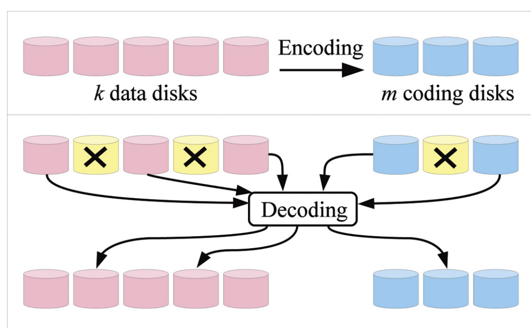


Figure 1: An erasure-coded storage system encodes k data disks onto m coding disks. When up to m disks fail, their contents are decoded by the erasure code.

coding disks, respectively. The acts of encoding and decoding are pictured in Figure 1.

With encoding, the contents of the k data disks are used to calculate the contents of the m coding disks. When up to m disks fail, their contents are decoded from the surviving disks. Repeating from above, when a disk fails, the failure mode is an erasure, where its contents are considered to be unreadable.

The simplest erasure codes assume that each disk holds one w-bit word. I label these words d_0, …, d_{k-1}, which are the data words stored on the data disks, and c_0, …, c_{m-1}, which are the coding words stored on the coding disks. The coding words are defined as linear combinations of the data words:

$$c_0 = a_{(0,0)}d_0 + \ldots + a_{(0,k-1)}d_{k-1}$$
$$c1 = a_{(1,0)}d_0 + \ldots + a_{(1,k-1)}d_{k-1}$$
$$\ldots$$
$$\ldots$$
$$c_{m-1} = a_{(m-1,0)}d_0 + \ldots + a_{(m-1,k-1)}d_{k-1}$$

The coefficients a are also w-bit words. Encoding, therefore, simply requires multiplying and adding words, and decoding involves solving a set of linear equations with Gaussian elimination or matrix inversion.

The arithmetic of erasure coding is special. When w=1, all of the d, c and a variables are single bits, and the arithmetic is standard arithmetic modulo 2: addition is binary XOR (⊕) and multiplication is binary AND. When w is larger, the arithmetic is called Galois Field arithmetic, denoted GF(2^w). This arithmetic operates on a closed set of numbers from 0 to 2^{w-1} in such a way that addition, multiplication, and division all have the properties that we expect. Conveniently, addition in a Galois Field is equal to bitwise XOR. Multiplication is more complicated, and beyond the scope of this article; however, there is a great deal of reference material on Galois Field arithmetic plus a variety of open source implementations (please see the annotated bibliography).

A disk, of course, holds more than a single w-bit word; however, with these simple codes, I partition each disk into w-bit words, and the i-th words on each disk are encoded and decoded together, independently of the other words. So that disks may be partitioned evenly into w-bit words, w is typically selected to be a power of two. Popular values are w=1 for its simplicity, because the arithmetic is composed of XORs and ANDs, and w=8, because each word is a single byte. In general, larger values of w allow for richer erasure codes, but the Galois Field arithmetic is more complex computationally.

An erasure code is therefore defined by w and the coefficients $a_{(i,j)}$. If the code successfully tolerates the failures of any m of the n disks, then the code is optimal with respect to fault-tolerance for the amount of extra space dedicated to coding. This makes sense, because one wouldn't expect to add m disks of redundancy and be able to tolerate more than m disk failures. If a code achieves this property, it is called *maximum distance separable* (MDS), a moniker that conveys zero intuition in a storage system. Regardless, MDS codes are desirable, because they deliver optimal fault tolerance for the space dedicated to coding.

In real storage settings, disks are partitioned into larger units called strips, and the set of corresponding strips from each of the n disks that encode and decode together is called a *stripe*. Each stripe is an independent entity for erasure coding, which allows the storage system designer to be flexible for a variety of reasons. For example, one may wish to rotate the identities of the n disks on a stripe-by-stripe basis, as in the left side of Figure 2. This is

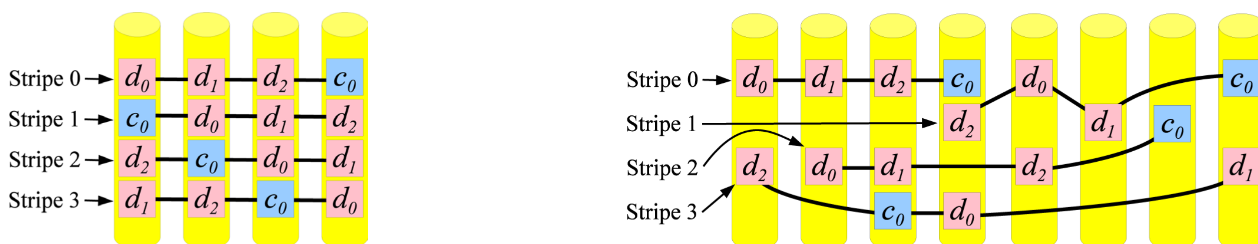


Figure 2: Two examples of laying out stripes on a collection of disks. On the left, there are n=4 disks, and each stripe contains k=3 strips of data and m=1 of coding. So that load is balanced, each stripe rotates the identities of the disks. On the right, there are now eight disks; however stripes still contain n=4 strips, three of which are data and one of which is coding.

a balanced approach, where each of the n=4 disks contains the same ratio of data and coding strips.

On the right side, a more ad hoc approach to laying out stripes is displayed. There are eight disks in this system; however, each stripe is composed of three data strips and one coding strip. Thus, the erasure code may be the same as in the left side of the figure; the allocation of strips to stripes is the only difference. This approach was used by Panasas to allow for flexible block allocation, and to allow additional disks to be added seamlessly to the storage system.

RAID-4 and RAID-5

Within this framework, I can define RAID-4 and RAID-5 as using the same simple erasure code, but having different stripe layouts. The code is an MDS code where m=1, w=1, and all of the a coefficients are also one. The sole coding bit is labeled p, and it is the XOR of all of the data bits:

$$p = d_0 \oplus d_1 \oplus \ldots \oplus d_{k-1}.$$

When any bit is erased, it may be decoded as the XOR of the surviving bits.

Although this equation operates on single bits, its implementation in a real system is extremely efficient, because whole strips may be XOR'd together in units of 128 or 256 bits using vector instructions such as Intel SSE2 (128 bits) or AVX (256 bits).

RAID-4 and RAID-5 both use the same erasure code; however, with RAID-4, the identity of each disk is fixed, and there is one disk, P, dedicated solely to coding. With RAID-5, the identities are rotated on a stripe-by-stripe basis as in the left side of Figure 2. Therefore, the system is more balanced, with each disk equally holding data and coding.

Linux RAID-6

RAID-6 systems add a second disk (called Q) to a RAID-4/5 system and tolerate the failure of any two disks. This requires an MDS erasure code where m=2, which is impossible to achieve with a simple XOR code. The solution implemented by the Red Hat Linux kernel employs the following simple code for w=8:

$$p = d_0 \oplus d_1 \oplus \ldots \oplus d_{k-1}$$
$$q = d_0 \oplus 2(d_1) \oplus \ldots \oplus 2^{k-1}(d_{k-1})$$

This code has some interesting properties. First, because addition in a Galois Field is equivalent to XOR, the P disk's erasure coding is equivalent to RAID-4/5. Second, the Q disk may be calculated using only addition and multiplication by two, because:

$$q = 2 (2 (\ldots 2 (2d_{k-1} \oplus d_{k-2}) \ldots) \oplus d_1) \oplus d0.$$

This is important because there are techniques to multiply 128- and 256-bit vectors of bytes by two in GF(28) with a small number of SSE/AVX instructions.

Reed-Solomon Codes

Reed-Solomon codes are MDS codes that exist whenever $n \le 2^w$. For example, so long as a storage system contains 256 disks or less, there is a Reed-Solomon defined for it that uses arithmetic in $GF(2^8)$. There are multiple ways to define the $a_{(i,j)}$ coefficients. The simplest to explain is the "Cauchy" construction: Choose n distinct numbers in $GF(2^w)$ and partition them into two sets X and Y such that X has m elements and Y has k. Then:

$$a_{(ij)} = \frac{1}{x_i \oplus y_j},$$

where arithmetic is over $GF(2^w)$.

Reed-Solomon codes are important because of their generality: they exist and are easy to define for any value of k and m. They have been viewed historically as expensive, because the CPU complexity of multiplication in a Galois Field is more expensive than XOR; however, vector instruction sets such as Intel SSE3 include operations that enable one to multiply 128-bit vectors of bytes by constants in a Galois Field with a small number of instructions. Although not as fast as multiplying by two as they do for a RAID-6 Q disk, it is fast enough that in most Reed-Solomon coding installations, disk I/O and even cache speeds are larger bottlenecks than the CPU. There are multiple open source libraries that implement Reed-Solomon coding for storage installations.

Array Codes

Array codes for storage systems arose in the 1990s. They were motivated by the desire to avoid Galois Field arithmetic and implement codes solely with the XOR operation. In the simple codes above, each disk logically holds one w-bit word, and thus there are m coding words, each of which is a different linear combination of the k data words. In an array code, each disk holds r w-bit words. Thus, there are mr coding words, each of which is a different linear combination of the kr data words.

They are called "array codes" because the coding system may be viewed as an r × n array of words, where the columns of the array are words that are co-located on the same disk. I depict an example in Figure 3. This is the RDP erasure code for k=4 and m=2. As such, it is a RAID-6 code. Each disk holds four bits, which means that r=4 and w=1. In the picture, I draw the array with the Q words on the left, the P words on the right, and the data words in the middle. The horizontal gray bars indicate XOR equations for the P disk's bits, and the other lines indicate how the Q disk's bits are encoded.

The allure of array codes for w=1 is that encoding and decoding require only XOR operations, yet the codes may be defined so that they are MDS. Examples are RDP, EVENODD, Blaum-Roth and Liberation codes for RAID-6, the STAR code for m=3, and Cauchy Reed-Solomon, Generalized EVENODD and Generalized RDP, which are defined for all values of k and m.

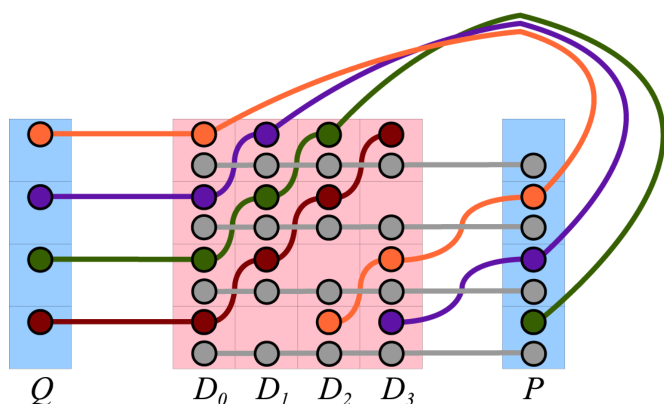


Figure 3: The RDP array code with the following parameters: k=4, m=2 (RAID-6), n = k+m = 6, r=4, w=1. The gray lines depict the coding equations for the P disk. The other lines depict the coding equations for the Q disk.

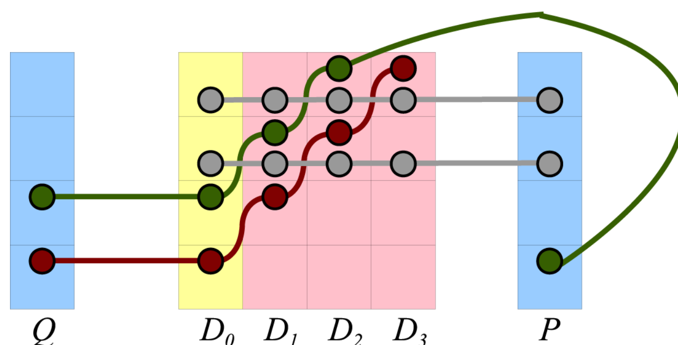


Figure 4: Recovering from a single failure in RDP. Only 12 bits are required, as opposed to 16 bits when one recovers solely from the P disk.

As mentioned in the section on mechanics, above, the advent of vector instructions has lowered the CPU burden of Galois Field arithmetic, and thus the allure of array codes has diminished in recent years; however, they have interesting properties with respect to recovery that make them viable alternatives to the simple codes. I explain this in the next section.

Recent Work #1: Reduced Disk I/O For Recovery

When one is using a simple erasure code and a single disk fails, the only way to recover its contents is to pick one of the m coding equations and use it to decode. This requires one to read k–1 strips from the surviving disks to calculate each strip on the failed disk. With an array code, one may significantly reduce the amount of data that is read from the surviving disks. I present an example in Figure 4, using the RDP code from Figure 3. In this example, the disk D_0 has failed and needs to be decoded. Were a simple erasure code employed, recovery would be equivalent to decoding solely from the P drive, where 16 bits must be read from the surviving disks; however, because of the structure of RDP, a judicious choice of decoding equations from both the P and Q drive allows one to decode D_0 by reading only 12 bits from the surviving disks.

As described in the section on mechanics, each bit in the description of the code corresponds to a larger block of storage on disk, which means that this example reduces the I/O costs of recovery by 25 percent. This observation was first made by Xiang in 2010, and further research has applied it to other array codes.

Recent Work #2: Regenerating Codes

Regenerating codes focus on reducing network I/O for recovery in distributed, erasure-coded storage systems. When one or more storage nodes fail, the system replaces them, either with nodes that hold their previous contents, or with nodes that hold equivalent contents from an erasure-coding perspective. In other words, the new collection of n nodes may hold different contents than the old collection; however, it maintains the property that the data may be calculated from any k of the nodes.

The calculation of these new nodes is performed so that network I/O is minimized. For example, suppose one storage node has failed and must be replaced. A simple erasure code requires k–1 of the other nodes to read their contents and send them for reconstruction. The schemes from Xiang (previous section) may leverage array codes so that more than k–1 nodes read and transmit data, but the total amount of data read and transmitted is reduced from the simple case. A properly defined regenerating code has the surviving nodes read even more data from disk, but then they massage it computationally so that they transmit even less data to perform regeneration.

Research on regenerating codes is both active and prolific. Please see the bibliography for summaries and examples.

Recent Work #3: Non-MDS Codes

A non-MDS code does not tolerate all combinations of m failures, and therefore the fault-tolerance is not optimal for the amount of extra storage committed to erasure coding; however, relaxation of the MDS property is typically accompanied by performance improvements that are impossible to achieve with MDS codes. I give a few examples here.

Flat XOR codes are simple codes where w=1. When m > 1, they are non-MDS; however, they have attractive features in comparison to their MDS counterparts. First, since w=1, they are based solely on the XOR operation—no Galois Field arithmetic is required. Second, they reduce both the I/O and the CPU complexity of encoding and decoding. When k and m grow to be very large (in the hundreds or thousands), flat XOR codes like Tornado and Raptor codes provide good degrees of fault-tolerance, while only requiring small, constant numbers of I/Os and XORs

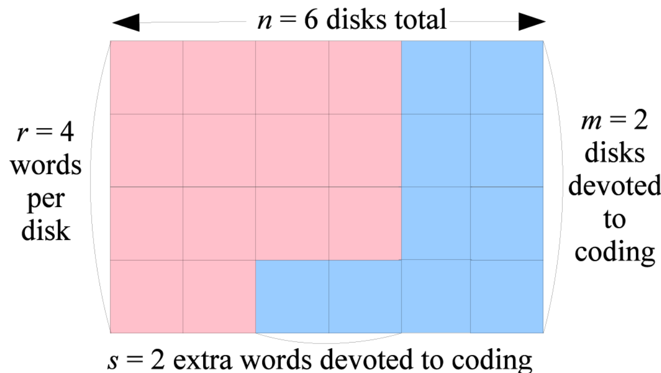


Figure 5: The layout of a stripe with an SD code, which tolerates the failure of any two disks and any additional two words in the stripe.

for encoding and decoding. This is as opposed to an MDS code, which necessarily requires O(k) I/Os and arithmetic operations. Other non-MDS codes that reduce complexity and rely solely on XOR are HoVeR, WEAVER, and GRID.

A second important class of non-MDS codes partitions the data words into groups, and divides the coding words into "local parities" and "global parities." Each local parity word protects a single group of data words, whereas each global parity word protects all of the words. The system is then fault-tolerant to a certain number of failures per data group, plus an additional number of failures for the entire system. The computational and I/O costs are smaller than an MDS system, yet the failure coverage is provably optimal for this coding paradigm. Examples of these codes are LRC codes that are implemented in Microsoft's Azure storage system, an identically named but different LRC code that has an open-source implementation in Hadoop, and Partial-MDS codes from IBM.

Finally, Sector-Disk (SD) codes are a class of non-MDS codes where m disks and s sectors per stripe are dedicated to fault-tolerance. An example is drawn in Figure 5, where a 6-disk system requires each disk to hold four words in its stripe. Two disks are devoted to fault-tolerance, and two additional words in the stripe are also devoted to fault-tolerance. The codes are designed so that they tolerate the failure of any two disks and any two additional words in the stripe. Thus, their storage overhead and fault-tolerance match the mixed failure modes of today's disks, where sector failures accumulate over time, unnoticed until a disk failure requires that they be read for recovery.

Conclusion

In this article, I have presented how erasure codes are leveraged by storage systems to tolerate the failure of disks, and in some cases, parts of disks. There are simple erasure codes, such as RAID-4/5 and Reed-Solomon codes, that view each disk as holding a single w-bit word, and define the coding words as linear combinations of the data words, using either XOR or Galois Field arithmetic. Array codes view each disk as holding multiple w-bit words, and achieve richer fault-tolerance, especially for codes based solely on the XOR operation. More recent work has focused on reducing the disk and network I/O requirements of the erasure codes, and on loosening the fault-tolerance requirements of the codes to improve performance.

Annotated Bibliography

In this section, I provide reference material for the various topics in the article. The following papers provide reference on implementing Galois Field arithmetic for erasure coding, including how to use vector instructions to accelerate performance drastically. The paper by Anvin [5] details the Linux RAID-6 implementation of Reed-Solomon coding.

[1] K. Greenan, E. Miller, and T. J. Schwartz. Optimizing Galois Field arithmetic for diverse processor architectures and applications. In MASCOTS 2008: 16th IEEE Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, Baltimore, MD, September 2008.

[2] J. Luo, K. D. Bowers, A. Oprea, and L. Xu. Efficient software implementations of large finite fields GF(2n) for secure storage applications. *ACM Transactions on Storage* 8(2), February 2012.

[3] 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.

[4] J. S. Plank, K. M. Greenan, and E. L. Miller. Screaming fast Galois Field arithmetic using Intel SIMD instructions. In FAST-2013: 11th USENIX Conference on File and Storage Technologies, San Jose, February 2013.

[5] H. P. Anvin. The mathematics of RAID-6: http://kernel.org/pub/linux/kernel/people/hpa/raid6.pdf, 2009.

[6] H. Li and Q. Huan-yan. Parallelized network coding with SIMD instruction sets. *International Symposium on Computer Science and Computational Technology,* IEEE, December 2008, pp. 364–369.

The following are open-source implementations of Galois Field arithmetic and erasure coding:

[7] Onion Networks. Java FEC Library v1.0.3. Open source code distribution: http://onionnetworks.com/fec/javadoc/, 2001.

[8] A. Partow. Schifra Reed-Solomon ECC Library. Open source code distribution: http://www.schifra.com/downloads.html, 2000-2007.

[9] J. S. Plank, K. M. Greenan, E. L. Miller, and W. B. Houston. GF-Complete: A comprehensive open source library for Galois

Field arithmetic. Technical Report UT-CS-13-703, University of Tennessee, January 2013.

[10] J. S. Plank, S. Simmerman, and C. D. Schuman. Jerasure: A library in C/C++ facilitating erasure coding for storage applications—Version 1.2. Technical Report CS-08-627, University of Tennessee, August 2008.

[11] L. Rizzo. Erasure codes based on Vandermonde matrices. Gzipped tar file posted: http://planete-bcast.inrialpes.fr/rubrique.php3?id_rubrique=10, 1998.

Besides my tutorial on Reed-Solomon coding for storage systems [3], the textbook by Peterson describes Reed-Solomon coding in a more classic manner. The papers by Blomer et al. and Rabin explain the "Cauchy" Reed-Solomon coding construction:

[12] J. Blomer, M. Kalfane, M. Karpinski, R. Karp, M. Luby, and D. Zuckerman. An XOR-based erasure-resilient coding scheme. Technical Report TR-95-048, International Computer Science Institute, August 1995.

[13] W. W. Peterson and E. J. Weldon, Jr. *Error-Correcting Codes, Second Edition.* The MIT Press, Cambridge, Massachusetts, 1972.

[14] M. O. Rabin. Efficient dispersal of information for security, load balancing, and fault tolerance. *Journal of the ACM* 36(2):335–348, April 1989.

The following papers describe array codes for RAID-6 that are based solely on the XOR operation:

[15] M. Blaum, J. Brady, J. Bruck, and J. Menon. EVENODD: An efficient scheme for tolerating double disk failures in RAID architectures. *IEEE Transactions on Computing* 44(2):192–202, February 1995.

[16] M. Blaum and R. M. Roth. On lowest density MDS codes. *IEEE Transactions on Information Theory* 45(1):46–59, January 1999.

[17] P. Corbett, B. English, A. Goel, T. Grcanac, S. Kleiman, J. Leong, and S. Sankar. Row diagonal parity for double disk failure correction. In FAST-2004: 3rd USENIX Conference on File and Storage Technologies, San Francisco, CA, March 2004.

[18] J. S. Plank, A. L. Buchsbaum, and B. T. Vander Zanden. Minimum density RAID-6 codes. *ACM Transactions on Storage* 6(4), May 2011.

Blomer et al.'s paper [12] describes how to convert a standard Reed-Solomon code into an array code that only uses XORs. The next three papers describe other general MDS array codes where w=1:

[19] M. Blaum, J. Bruck, and A. Vardy. MDS array codes with independent parity symbols. *IEEE Transactions on Information Theory* 42(2):529—542, February 1996.

[20] M. Blaum. A family of MDS array codes with minimal number of encoding operations. In IEEE International Symposium on Information Theory, Seattle, September 2006.

[21] C. Huang and L. Xu. STAR: An efficient coding scheme for correcting triple storage node failures. *IEEE Transactions on Computers* 57(7):889–901, July 2008.

The following papers reduce the amount of data that must be read from disk when performing recovery on XOR-based array codes:

[22] O. Khan, R. Burns, J. S. Plank, W. Pierce, and C. Huang. Rethinking erasure codes for cloud file systems: Minimizing I/O for recovery and degraded reads. In FAST-2012: 10th USENIX Conference on File and Storage Technologies, San Jose, February 2012.

[23] Z. Wang, A. G. Dimakis, and J. Bruck. Rebuilding for array codes in distributed storage systems. In *GLOBECOM ACTEMT Workshop,* pp. 1905–1909. IEEE, December 2010.

[24] L. Xiang, Y. Xu, J. C. S. Lui, and Q. Chang. Optimal recovery of single disk failure in RDP code storage systems. In ACM SIGMETRICS, June 2010.

The following papers summarize and exemplify research on regenerating codes:

[25] V. Cadambe, C. Huang, J. Li, and S. Mehrotra. Compound codes for optimal repair in MDS code based distributed storage systems. In Asilomar Conference on Signals, Systems and Computers, 2011.

[26] A. G. Dimakis, P. B. Godfrey, Y. Wu, M. Wainwright, and K. Ramchandran. Network coding for distributed storage systems. *IEEE Transactions on Information Theory,* 2010.

[27] A. G. Dimakis, K. Ramchandran, Y. Wu, and C. Suh. A survey on network codes for distributed storage. *Proceedings of the IEEE* 99(3), March 2011.

The following papers describe XOR-based, non-MDS codes that improve the performance of encoding and recovery:

[28] K. M. Greenan, X. Li, and J. J. Wylie. Flat XOR-based erasure codes in storage systems: Constructions, efficient recovery and tradeoffs. In 26th IEEE Symposium on Massive Storage Systems and Technologies (MSST2010), Nevada, May 2010.

[29] J. L. Hafner. WEAVER Codes: Highly fault tolerant erasure codes for storage systems. In *FAST-2005: 4th USENIX Conference on File and Storage Technologies,* pp. 211–224, San Francisco, December 2005.

[30] J. L. Hafner. HoVer erasure codes for disk arrays. In DSN-2006: The International Conference on Dependable Systems and Networks, Philadelphia, June 2006.

[31] M. Li, J. Shu, and W. Zheng. GRID codes: Strip-based erasure codes with high fault tolerance for storage systems. *ACM Transactions on Storage* 4(4), January 2009.

[32] M. Luby, M. Mitzenmacher, A. Shokrollahi, D. Spielman, and V. Stemann. Practical loss-resilient codes. In *29th Annual ACM Symposium on Theory of Computing,* pages 150–159, El Paso, TX, 1997. ACM.

[33] A. Shokrollahi. Raptor codes. *IEEE Transactions on Information Theory,* pages 2551–2567, 2006.

The following papers describe non-MDS erasure codes that feature local and global parity words and address cloud storage systems or mixed failure modes in RAID systems:

[34] M. Blaum, J. L. Hafner, and S. Hetzler. Partial-MDS codes and their application to RAID type of architectures. *IEEE Transactions on Information Theory,* July 2013.

[35] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, and S. Yekhanin. Erasure coding in Windows Azure storage. In USENIX Annual Technical Conference, Boston, June 2012.

[36] J. S. Plank, M. Blaum, and J. L. Hafner. SD codes: Erasure codes designed for how storage systems really fail. In FAST-2013: 11th USENIX Conference on File and Storage Technologies, San Jose, February 2013.

[37] M. Sathiamoorthy, M. Asteris, D. Papailiopoulos, A. G. Dimakis, R. Vadali, S. Chen, and D. Borthakur. XORing elephants: Novel erasure codes for big data. In 39th International Conference on Very Large Data Bases, August 2013.

Acknowledgements

The author thanks Rik Farrow for encouraging him to write this article. This material is based upon work supported by the National Science Foundation under grant CSR-1016636, and by an IBM Faculty Research Award. The author is indebted to his various erasure-coding cohorts and co-authors through the years: Mario Blaum, Randal Burns, Kevin Greenan, Jim Hafner, Cheng Huang, Ethan Miller, Jason Resch, Jay Wylie, and Lihao Xu.

Professors, Campus Staff, and Students—do you have a USENIX Representative on your campus? If not, USENIX is interested in having one!

The USENIX Campus Rep Program is a network of representatives at campuses around the world who provide Association information to students, and encourage student involvement in USENIX. This is a volunteer program, for which USENIX is always looking for academics to participate. The program is designed for faculty who directly interact with students. We fund one representative from a campus at a time. In return for service as a campus representative, we offer a complimentary membership and other benefits.

A campus rep's responsibilities include:

- Maintaining a library (online and in print) of USENIX publications at your university for student use
- Distributing calls for papers and upcoming event brochures, and re-distributing informational emails from USENIX
- Encouraging students to apply for travel grants to conferences
- Providing students who wish to join USENIX with information and applications
- Helping students to submit research papers to relevant USENIX conferences
- Providing USENIX with feedback and suggestions on how the organization can better serve students

In return for being our "eyes and ears" on campus, representatives receive a complimentary membership in USENIX with all membership benefits (except voting rights), and a free conference registration once a year (after one full year of service as a campus rep).

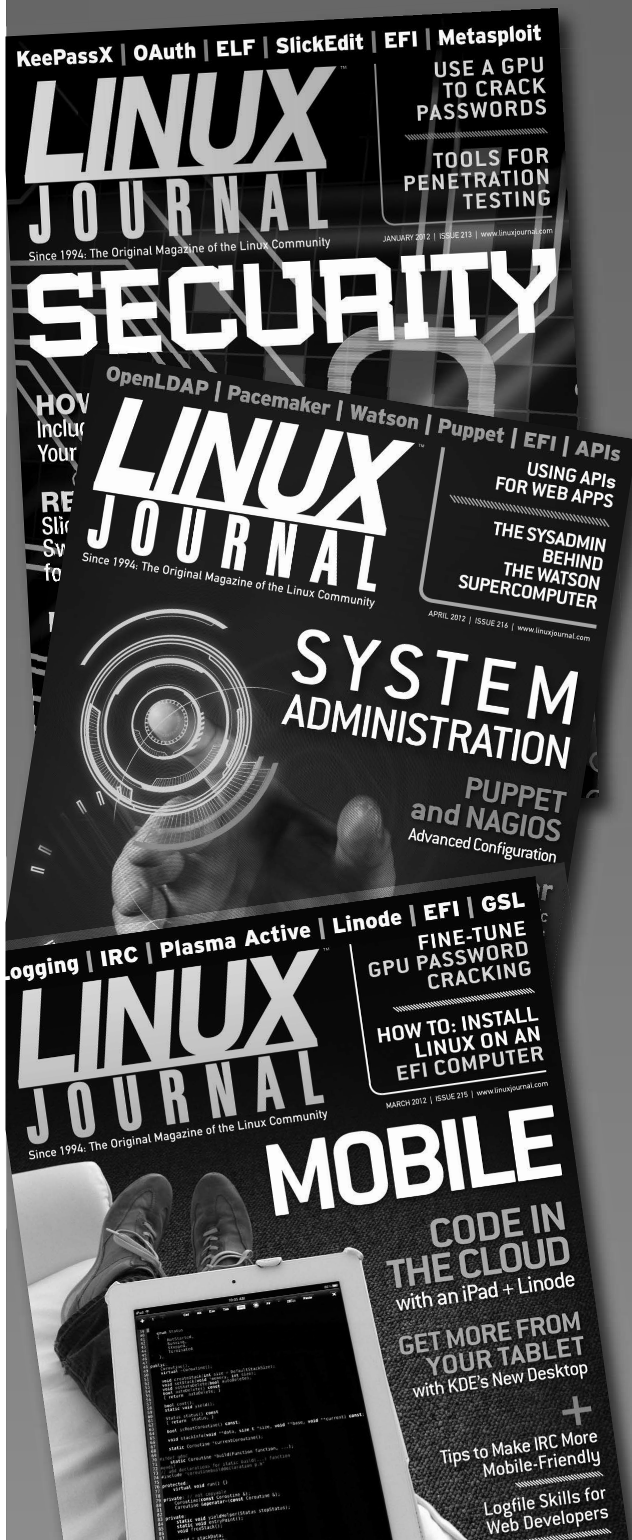
To qualify as a campus representative, you must:

- Be full-time faculty or staff at a four year accredited university
- Have been a dues-paying member of USENIX for at least one full year in the past

For more information about our Student Programs, contact
Julie Miller, Marketing Communications Manager, julie@usenix.org

www.usenix.org/students

If You Use Linux, You Should Be Reading LINUX JOURNAL™



- In-depth information providing a full 360-degree look at featured topics relating to Linux
- Tools, tips and tricks you will use today as well as relevant information for the future
- Advice and inspiration for getting the most out of your Linux system
- Instructional how-tos will save you time and money

Subscribe now for instant access! For only $29.50 per year—less than $2.50 per issue—you'll have access to *Linux Journal* each month as a PDF, in ePub & Kindle formats, on-line and through our Android & iOS apps. Wherever you go, *Linux Journal* goes with you.

SUBSCRIBE NOW AT:

WWW.LINUXJOURNAL.COM/SUBSCRIBE