A Performance Comparison of Open-Source Erasure Coding Libraries for Storage Applications

Catherine D. Schuman James S. Plank

Technical Report UT-CS-08-625
Department of Electrical Engineering and Computer Science
University of Tennessee

August 8, 2008

[cschuman,plank]@cs.utk.edu

The home for this paper is http://www.cs.utk.edu/~plank/plank/papers/CS-08-625.html. The current version of this paper is not intented for publication, but to provide the seeds for collaboration with other erasure code developers and users.

Abstract

Erasure coding is a fundemental technique to prevent data loss in storage systems composed of multiple disks. Recently, there have been multiple open-source implementations of a variety of erasure codes. In this work, we present a comparison of the performance of various codes and implementations, concentrating on encoding and decoding. It is hard to draw overarching conclusions from a single performance study. However, performance data is important to gain an understanding of the real-life performance ramifications of code properties and implementation decisions. The significance of this paper is to guide those who use and design codes, so that they may be able to predict what performance to expect when using an erasure code. One important, although obvious, conclusion is that reducing cache misses is more important than reducing XOR operations.

1 Introduction

In recent years, erasure codes have moved to the fore to prevent data loss in storage systems composed of multiple disks. Storage companies such as Cleversafe [6], Data Domain [32], Network Appliance [15] and Panasas [27] are delivering products that use erasure codes for data availability. Academic projects such as Oceanstore [24], LoCI [2] and Pergamum [26] are doing the same. And equally important, major technology corporations such as HP [29], IBM [8, 9] and Microsoft [11, 12] are performing active research on erasure codes for storage systems.

Along with proprietary implementations of erasure codes, there have been numerous open source implementations of a variety of erasure codes that are available for download [6, 13, 16, 18, 28]. While one cannot expect the same level of performance and functionality from an open source implementation that one gets with a proprietary implementation, it is the intent of some of these projects to provide storage system developers with high quality tools. As such, there is a need to understand how these codes and implementations perform.

In this paper, we seek to gain such an understanding. We have installed all of the above erasure coding implementations on machines at the University of Tennessee, and have performed two bench-

marking tests on them. The first performs a fundamental operation of taking a large file (in this case, a video file), breaking it up into slices for multiple storage nodes and encoding it into other slices for other storage nodes. The second performs another fundamental operation which is to recompute the contents of the file from remaining the slices when some of the slices have failed.

The conclusions that we draw are somewhat obvious. The coding technique, implementation and word size all affect the performance of coding. There is a great deal of variation in encoding and decoding performance of multiple implementations of the same coding technique. For standard Reed-Solomon coding, the "Zfec" implementation [28] performs the fastest. The Cauchy Reed-Solomon code implementation of the **Jerasure** library [18], which optimizes the encoding matrices, vastly outperforms the other implementations.

In any performance study, effects due to the memory hierarchy must be observed, and a final experiment demonstrates clearly that the encoding parameters should take account of the machine's cache size to achieve best performance.

2 Nomenclature and Basic Erasure Coding

It is an unfortunate consequence of the history of erasure coding research that there is no unified nomenclature for erasure coding. We borrow terminology mostly from Hafner *et al* [10], but try to conform to more classic coding terminology (e.g. [4, 14]) when appropriate.

Our storage system is composed of an **array** of n disks, each of which is the same size. Of these n disks, k of them hold **data** and the remaining m hold **coding** information, often termed **parity**. We label the data disks $D_0, \ldots, D_{k-1}$ and the parity disks $C_0, \ldots, C_{m-1}$. An example system is pictured in Figure 1.

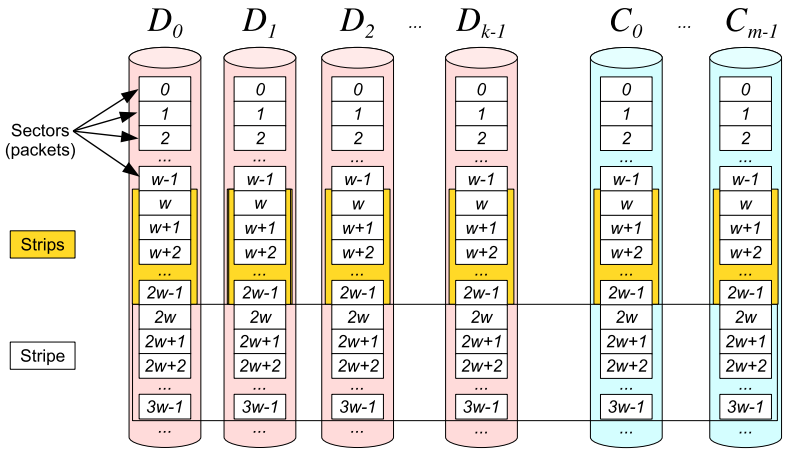

Figure 1: An example storage system with erasure coding. There are k data disks and m coding disks.

Disks are composed of **sectors**, which are the smallest units of I/O to and from a disk. Sector sizes typically range from 512 to 4096 bytes and are always powers of two. Most erasure codes are defined such that disks are logically organized to hold bits; however, when they are implemented, all the bits on one sector are considered as one single entity called an **element**. When a code specifies that bit on drive C_0 equals the exclusive-or of bits on drives D_0 and D_1, that really means that a sector on drive C_0 will be calculated to be the parity (bitwise exclusive-or) of corresponding sectors on drives D_0 and D_1. We will thus use the terms *bit*, *sector* and *element* interchageably in this paper. We also use the term **packet** to denote a unit logically equivalent to a sector, but that can be much bigger.

A code typically has a **word size** w, which means that for the purposes of coding, each disk is partitioned into **strips** of w sectors each. Each of these strips is encoded and decoded independently

from the other strips on the same disk. The collection of strips from all disks on the array that encode and decode together is called a **stripe**.

Thus, to define an erasure code, one must specify the word size w, and then one must describe the way in which sectors on the parity strips are calculated from the sectors on the data strips in the same stripe.

For small values of m, **Maximum Distance Seperable (MDS)** codes are highly desirable. These are codes having the property that when the number of failed disks in the is less than or equal to m, the original data can always be reconstructed. In this work, we will only consider MDS codes. We also only consider **horizontal** codes, where disks hold exclusively data or parity, rather than **vertical** codes (e.g. [9, 30, 31]) that require disks to hold both.

2.1 Standard MDS Codes

The grandfather of erasure codes is the set of **Reed-Solomon codes** [23]. These are general purpose MDS codes that may be constructed for any values of k and m. Their classic presentation is for noisy communication channels, where serial data must be transmitted in a way that tolerates arbitrary (and often bursty) bit flips. These are denoted **errors**. Storage systems do not fail with errors, but with **erasures**, where entire disks or sectors become unusable. Erasures differ from errors in that their failures are identifiable. Thus the treatment of Reed-Solomon coding for storage differs somewhat from its classical treatment. Tutorial instructions for employing Reed-Solomon codes for storage systems are available [17, 21], and there are several open-source implementations [16, 18, 28].

The crux of Reed-Solomon coding is to use linear algebra. In particular, encoding is defined by a matrix-vector product where the vector is composed of k words of data, and the product is composed of m words of encoded data. The matrix is called a **generator** matrix, although some sources call it a "distribution" matrix. The words are w bits long, where $k+m \leq 2^w$. A special kind of arithmetic, termed **Galois Field** arithmetic, $GF(2^w)$, is used to perform the matrix-vector product. In this arithmetic, addition is equal to bitwise exclusive-or (XOR), and multiplication/division may be implemented in a variety of ways, all of which are much more expensive than XOR.

Classic Reed-Solomon coding systems do not follow Figure 1, but instead treat each disk as a collection of words, and encode each set of words independently from each other set. As such, it is extremely convenient to have words fall on machine word boundaries, meaning that for classic Reed-Solomon coding, w is restricted to be 8, 16, 32 or 64.

In 1995, Blomer *et al* presented **Cauchy Reed-Solomon (CRS) codes** [5]. The major difference with CRS codes is that they expand the generator matrix by a factor of w in each dimension, converting each matrix element into a $w \times w$ bit matrix. Additionally, they split the multiplicand and product vectors into k and m sets of w **packets**. This allows the product to be computed using only XOR operations, and it also frees w from the restriction of having to align on word boundaries. With CRS coding, the system looks exactly as in Figure 1, with each strip consisting of w packets. Stripes are each encoded and decoded independently.

The performance of CRS depends directly on the number of ones in the expanded generator matrix. In 2006, Plank and Xu made the observation that there are multiple ways of constructing the generator matrix for CRS, and that significant savings may be achieved by choosing "Good" matrices [22]. The **Jerasure** library extends this by massaging the matrix further [18]. In this work, we will refer to ***Original*** matrices, which are those defined by Blomer *et al*, and ***Good*** matrices, which are the improved matrices derived in the **Jerasure** library. Besides the **Jerasure** library, CRS coding has been implemented in open-source form by Luby [13] and by Cleversafe [6]. Both use *Original* matrices for coding.

2.2 RAID-6 Codes

RAID-6 systems are restricted storage systems for which $m = 2$. In a RAID-6 system, the two parity drives are called P and Q. Currently, most commercial storage systems that implement high

availability use RAID-6 [15, 27, 32]. There are several specialized coding algorithms for RAID-6. First, there is a clever tweak to Reed-Solomon coding that improves encoding performance for RAID-6 [1]. Next, there are two algorithms for RAID-6 that make use of "diagonal" parity construtions. These are **EVENODD** [3] and **RDP** [7] codes. RDP in particular features optimal encoding and decoding performance for certain values of k. Both codes are laid out as in Figure 1, where w must be a number such that $w + 1$ is prime and either $k \leq w + 1$ (EVENODD) or $k \leq w$ (RDP).

A final class of RAID-6 codes are **Minimal Density** codes, which work using an generator bitmatrix like CRS coding. However, unlike CRS coding, these matrices have a provably minimal number of ones. Three different constructions of Minimal Density exist: **Liberation** codes [20] for when w is prime, **Blaum-Roth** codes [4] for when $w + 1$ is prime, and the **Liber8tion** code [19] for $w = 8$. In all cases, $k \leq w$. The theoretical performance of minimal density codes lies between RDP and EVENODD when k is near w. It outperforms both when k is much smaller than w.

Finally, techniques to optimize the calculation of the bitmatrix-vector product have been explored by several researchers [12, 10, 20]. The **Jerasure** library implements the "Code-specific Hybrid Reconstruction" algorithm of Hafner *et al* [10], which can drastically reduce the number of XORs required for CRS encoding/decoding, and for decoding when using the minimal density techniques [19, 20].

3 Description of Open Source Libraries Tested

We used the following open source erasure coding libraries for our tests.

LUBY: CRS coding was developed by Luby *et al* in 1993-1994 for use in the PET project at ICSI. It were developed to be fast enough to run on real-time on small blocks. The **Luby** library is written C and is available for free download [13].

SCHIFRA: In July, 2000, the first version of the **Schifra** library was made avaiable for free download. The library is written in C++, with a robust API and support [16]. There is a license for documentation of the library and for a high-performance version. Thus, the version we have tested does not represent the best performance of **Schifra**. However, it does represent the performance a developer can expect from the freely available download.

ZOOKO: The "Zfec" library for erasure coding has been in development since 2004. The latest version (1.4.0) was posted in January, 2008. The library is programmable and portable. It includes command-line tools and APIs in C, Python and Haskell [28]. After evaluating several coding methodologies, the authors based their implementation of a well-known implementation of classic Reed-Solomon coding by Rizzo [25].

JERASURE: In September, 2007, the first version of the **Jerasure** library was made available [18]. The library is in C, and its intent is to be very flexible, implementing a wide variety of coding techniques and support for coding. Standard Reed-Solomon coding is implemented, along with CRS coding, including the improved the generator matrices, and the minimal density RAID-6 codes. It is released under the GNU LGPL.

CLEVERSAFE: In May, 2008, Cleversafe exported the first open source version of its dispersed storage system [6]. Written entirely in Java, it supports the same API as Cleversafe's proprietary system, which is notable as it is the first commercial distributed storage system to implement availability beyond RAID-6. For this paper, J. Resch of Cleversafe stripped out the erasure coding part of the open source distribution. It is based on Luby's original CRS implementation.

EVENODD/RDP: The implementation of EVENODD and RDP coding is a private version developed by Plank for comparison with Liberation codes [20]. Since EVENODD and RDP codes are patented, this implementation is not available to the public, as its sole intent is for performance comparison.

4 Main Experimental Setup

The scenario that we wish to explore is this: Suppose one wants to take a large file and distribute it to $n = k + m$ storage servers so that it may be reconstructed when up to m of the servers are down or unavailable. We explore this scenario with two experimental tests.

The first test is an encoding test. With this test, we take a large video file (736 MB), divide it into k data files and encode it into m coding files using the various methods, libraries, and word sizes. Each of the data and coding files is of size $736/k$ MB. We do not perform the act of spreading the various files among storage servers, since that activity has the same performance regardless of the coding method or implementation.

To test the performance, we implemented a dummy program that performs all of the file activities (reading the video file and creating the output files) but performs no coding — the m coding files simply contain zeroes. We subtract the running time of this program from the running time of the other encoders to remove the time for file activities and localize the performance of encoding. We then present the performance of encoding as the rate in which all encoding was completed.

For example, suppose $k = 6$ and $m = 2$. Our dummy implementation breaks the video file into six data files of 122.6 MB each, and creates two coding files, also of size 122.6 MB, which contain all zeros. On average, this takes 53.8 seconds. Now, suppose we wish to test the performance of Cleversafe's implementation of CRS coding with $w = 8$. We run our encoding program, which creates the same six data files and two coding files (which actually encode this time). This takes 90.3 seconds on average. Thus the encoding portion of the test takes $39.5 = 90.3 - 53.8$ seconds, and we report the performance to be $736/39.5 = 18.7$ MB/sec.

The second test is a decoding test, where we randomly delete m of the $n = k + m$ files that we created in encoding and reconstruct them from the remaining k pieces. As with encoding, we use a dummy program to localize the performance of decoding, and record the performance in MB/sec.

The parameters of the main experiment include the erasure coding technique, the library and the word size (w). All of the libraries except for **Jerasure** implement only one coding technique, and all except **Luby** and **Jerasure** have one preset word size. **Luby** and **Jerasure** also allow the programmer to set the packet size. After some initial performance tests, we selected a value of 1KB for the packet size in these tests. **Cleversafe** selects an "optimum slice size" based on the other parameters.

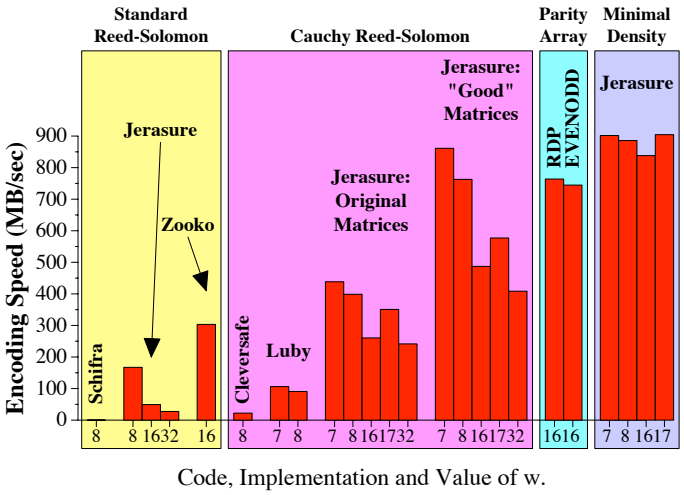

Figure 2: Encoding performance of a RAID-6 system with $k = 6$ ($m = 2$).

Finally, the encoder and decoder allow for the reading and writing to be performed in stages, with a fixed size buffer in memory. This is to avoid having to store the entire video file and coding information

in memory.

The machine for testing has a 2.6 GHz Intel Pentium 4 processor, 2 GB of RAM and a cache size of 512 KB. It runs Debian Linux, version 3.1. Each data point in the graphs that follow is the average of five runs.

5 Results from the Main Experiment

We present results from four sets of experiments. These are two RAID-6 tests ($k = 6$ and $k = 14$), plus two higher availability tests where the total number of disks numbers 16. These are $\{k = 12, m = 4\}$ and $\{k = 10, m = 6\}$. The latter set of parameters is the default set used by Cleversafe in their commercial storage system.

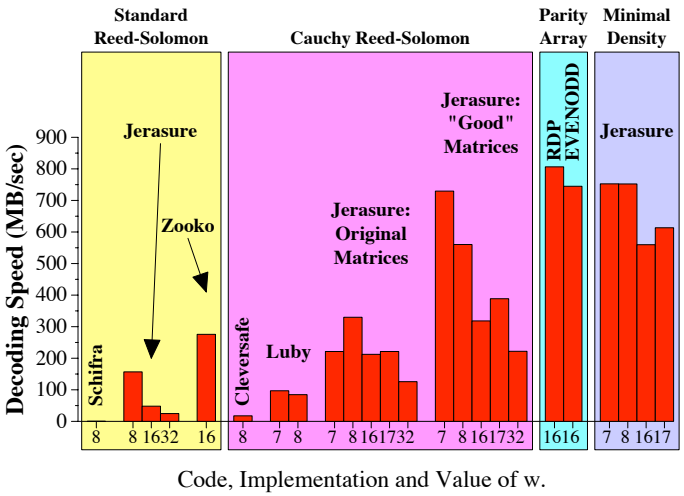

Figure 3: Decoding performance of a RAID-6 system with $k = 6$ ($m = 2$).

Figures 2 and 3 show the performance of the various implementations, codes and word sizes on a RAID-6 system with six data disks. The values of w are ones that are likely to be used in practice. For the minimal density codes, Liberation codes are employed for $w \in \{7, 17\}$, the Blaum-Roth code is used for $w = 16$, and the Liber8tion code is used for $w = 8$.

The most glaring feature of these graphs is the extremely wide range of performances, from under 50 MB/sec for **Schifra** and **Cleversafe**, to roughly 900 MB/sec for the minimal density codes. Of course, all parameters impact perforamnce — code type, implementation and word size. However, it is interesting that some *de facto* assumptions that are made in the community, for example that CRS always outperforms standard Reed-Solomon coding, clearly are not true.

Of the standard Reed-Solomon coding implementations, **Zooko** performs better than the others by a factor of two at the smallest. The **Schifra** library performs the slowest, which may be attributed to the fact that its open source library is meant to demonstrate functionality and not performance.

The CRS implemtations display a very high range in performance, with the **Jerasure** implementation performing the best. The generator matrix clearly has an important effect on performance – in **Jerasure**, the *good* matrices perform roughly a factor of two better than their *original* counterparts.

With regard to the specialized RAID-6 codes, it is surprising that the minimal density codes outperform RDP and EVENODD in encoding. We surmise that this is due to cache effects, or perhaps because the **Jerasure** code is meant to be exported and used by others, while the RDP/EVENODD implementations were written for a paper evaluation. The performance of minimal density decoding is worse than encoding, as expected [19, 20].

We include the remainder of the performance results in the Appendix. They mirror the results above. The performance of minimal density decoding when $k = 14$ and $m = 2$ is much worse than encoding; however, this is to be expected as k grows [20].

6 Packet and Cache Size Ramifications

Figuring out the sources of performance improvements and penalties is a difficult task on a real machine and application. As the graphs from section 5 show, counting XORs alone is not enough. Understanding the caching behavior of a program is difficult, but to a coarse degree, we can modify a code's packet size in an attempt to optimize cache behavior. We performed two brief experiments to assess this effect at a high level. In Figure 4, we test the effect of modifying the packet size on the encoding rate in four scenarios:

1. $k = 14$, $m = 2$, $w = 16$, Blaum-Roth coding.
2. $k = 14$, $m = 2$, $w = 7$, CRS coding (*good* matrix).
3. $k = 10$, $m = 6$, $w = 7$, CRS coding (*good* matrix).
4. $k = 10$, $m = 6$, $w = 16$, CRS coding (*good* matrix).

All use the **Jerasure** implementation, but plot only one or two runs per data point. The left graph shows an exploration of small packet sizes and the right graph shows an exploration of large packet sizes.

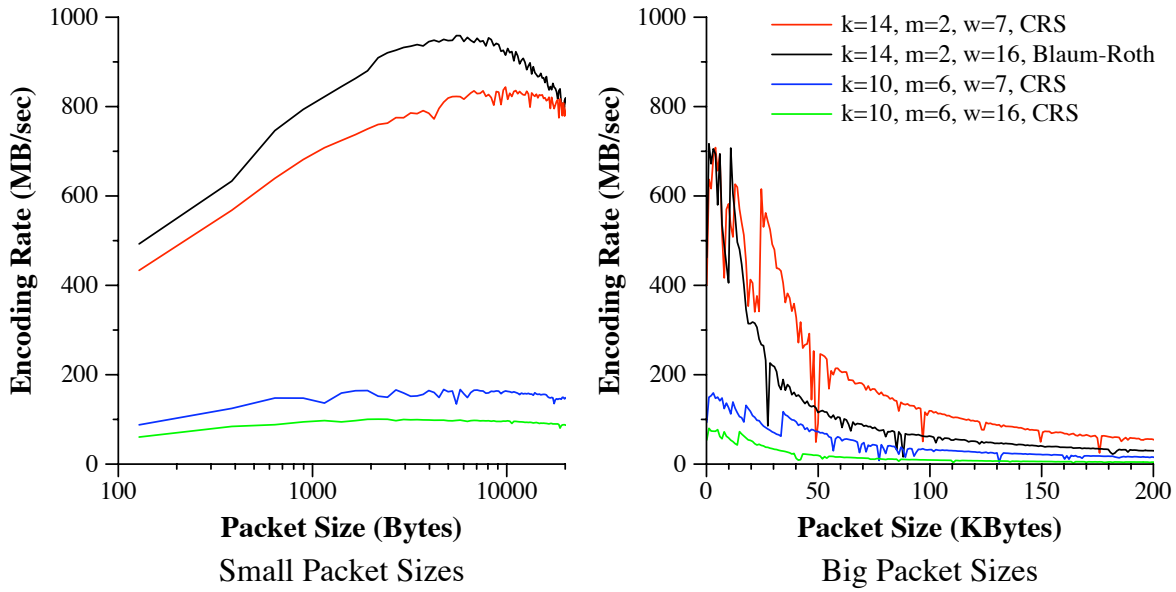

Figure 4: The effect of modifying the packet size in four coding scenarios.

As the graphs show, the effect of modifying the packet size is drastic. When packets are very small, the performance suffers because the inner loops peforming XOR operations is too small. As such, increasing the packet size improves performance to a point where the overhead of cache misses starts to penalize performance. As the right graph shows, this effect continues until the performance is nearly an order of magnitude worse than the best performance. This underscores the importance of considering caching behavior when performing erasure coding.

For a given packet size, a larger value of w means a larger stripe size. Accordingly, the stripe sizes in Figure 4 for $w = 16$ are $\frac{16}{7}$ larger than those for $w = 7$. This effect is reflected in the left graph of Figure 4, where the peak performance of the two codes when $w = 16$ is achieved at a smaller packet size than the codes for $w = 7$.

We acknowledge that this experiment is sketchy at best – more data needs to be taken, and more experimentation on this topic needs to be performed.

7 Concluding Remarks

This paper has compared the performance of several open source erasure coding libraries. Their performance runs the gamut from slow to fast, with factors being the erasure coding technique used, optimization of the underlying coding structure (i.e. the generator matrix in CRS coding), and attention to cache behavior.

While this work should be useful in helping storage practitioners evaluate and use erasure coding libraries, it is clear that more work can and should be done. First, the tests should be performed on multiple machines so that individual machine quirks do not impact results. Second, in the XOR codes, one may schedule the individual XOR operations in an exponential number of ways – doing so to improve cache utilization may yield further improvements in performance. It will be a challenge to perform this additional work, yet digest the results in a coherent way. We look forward to that challenge.

8 Acknowledgements

This material is based upon work supported by the National Science Foundation under grant CNS-0615221. The authors gratefully acknowledge Ilya Volvovski and Jason Resch from Cleversafe for so generously providing the erasure coding core of Cleversafe's open source implementation so that we could integrate it into our environment. Additionally, the authors acknowledge Lihao Xu and Jianqiang Luo for helpful interaction, and Zooko Wilcox-O'Hearn for encouraging communication over the past five years.

References

[1] Anvin, H. P. The mathematics of RAID-6. `http://kernel.org/pub/linux/kernel/people/hpa/raid6.pdf`, 2007.

[2] Beck *et al*, M. Logistical computing and internetworking: Middleware for the use of storage in communication. In *Third Anual International Workshop on Active Middleware Services (AMS)* (San Francisco, August 2001).

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

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

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

[6] Cleversafe, Inc. Cleversafe Dispersed Storage. Open source code distribution: `http://www.cleversafe.org/downloads`, 2008.

[7] Corbett, P., English, B., Goel, A., Grcanac, T., Kleiman, S., Leong, J., and Sankar, S. Row diagonal parity for double disk failure correction. In *4th Usenix Conference on File and Storage Technologies* (San Francisco, CA, March 2004).

[8] Hafner, J. L. HoVer erasure codes for disk arrays. Tech. Rep. RJ10352 (A0507-015), IBM Research Division, July 2005.

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

[10] Hafner, J. L., Deenadhayalan, V., Rao, K. K., and Tomlin, A. Matrix methods for lost data reconstruction in erasure codes. In *FAST-2005: 4th Usenix Conference on File and Storage Technologies* (San Francisco, December 2005), pp. 183–196.

[11] Huang, C., Chen, M., and Li, J. Pyramid codes: Flexible schemes to trade space for access efficienty in reliable data storage systems. In *NCA-07: 6th IEEE International Symposium on Network Computing Applications* (Cambridge, MA, July 2007).

[12] Huang, C., Li, J., and Chen, M. On optimizing XOR-based codes for fault-tolerant storage applications. In *ITW'07, Information Theory Workshop* (Tahoe City, CA, September 2007), IEEE, pp. 218–223.

[13] Luby, M. Code for Cauchy Reed-Solomon coding. Uuencoded tar file: `http://www.icsi.berkeley.edu/~luby/cauchy.tar.uu`, 1997.

[14] MacWilliams, F. J., and Sloane, N. J. A. *The Theory of Error-Correcting Codes, Part I.* North-Holland Publishing Company, Amsterdam, New York, Oxford, 1977.

[15] Nisbet, B. FAS storage systems: Laying the foundation for application availability. Network Appliance white paper: `http://www.netapp.com/us/library/analyst-reports/ar1056.html`, February 2008.

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

[17] Plank, J. S. A tutorial on Reed-Solomon coding for fault-tolerance in RAID-like systems. *Software – Practice & Experience 27*, 9 (September 1997), 995–1012.

[18] Plank, J. S. Jerasure: A library in C/C++ facilitating erasure coding for storage applications. Tech. Rep. CS-07-603, University of Tennessee, September 2007.

[19] Plank, J. S. A new minimum density RAID-6 code with a word size of eight. In *NCA-08: 7th IEEE International Symposium on Network Computing Applications* (Cambridge, MA, July 2008).

[20] Plank, J. S. The RAID-6 Liberation codes. In *FAST-2008: 6th Usenix Conference on File and Storage Technologies* (San Jose, February 2008), pp. 97–110.

[21] Plank, J. S., and Ding, Y. Note: Correction to the 1997 tutorial on Reed-Solomon coding. *Software – Practice & Experience 35*, 2 (February 2005), 189–194.

[22] Plank, J. S., and Xu, L. Optimizing Cauchy Reed-Solomon codes for fault-tolerant network storage applications. In *NCA-06: 5th IEEE International Symposium on Network Computing Applications* (Cambridge, MA, July 2006).

[23] Reed, I. S., and Solomon, G. Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics 8* (1960), 300–304.

[24] Rhea, S., Wells, C., Eaton, P., Geels, D., Zhao, B., Weatherspoon, H., and Kubiatowicz, J. Maintenance-free global data storage. *IEEE Internet Computing 5*, 5 (2001), 40–49.

[25] Rizzo, L. Effective erasure codes for reliable computer communication protocols. *ACM SIGCOMM Computer Communication Review 27*, 2 (1997), 24–36.

[26] Storer, M. W., Greenan, K. M., Miller, E. L., and Voruganti, K. Pergamum: Replacing tape with energy efficient, reliable, disk-based archival storage. In *FAST-2008: 6th Usenix Conference on File and Storage Technologies* (San Jose, February 2008), pp. 1–16.

[27] Welch, B., Unangst, M., Abbasi, Z., Gibson, G., Mueller, B., Small, J., Zelenka, J., and Zhou, B. Scalable performance of the panasas parallel file system. In *FAST-2008: 6th Usenix Conference on File and Storage Technologies* (San Jose, February 2008), pp. 17–33.

[28] Wilcox-O'Hearn, Z. Zfec 1.4.0. Open source code distribution: `http://pypi.python.org/pypi/zfec`, 2008.

[29] Wylie, J. J., and Swaminathan, R. Determining fault tolerance of XOR-based erasure codes efficiently. In *DSN-2007: The International Conference on Dependable Systems and Networks* (Edinburgh, Scotland, June 2007), IEEE.

[30] Xu, L., Bohossian, V., Bruck, J., and Wagner, D. Low density MDS codes and factors of complete graphs. *IEEE Transactions on Information Theory 45*, 6 (September 1999), 1817–1826.

[31] Xu, L., and Bruck, J. X-Code: MDS array codes with optimal encoding. *IEEE Transactions on Information Theory 45*, 1 (January 1999), 272–276.

[32] Zhu, B., Li, K., and Patterson, H. Avoiding the disk bottleneck in the Data Domain deduplication file system. In *FAST-2008: 6th Usenix Conference on File and Storage Technologies* (San Jose, February 2008), pp. 269–282.

Appendix: Performance Results for the other Parameters

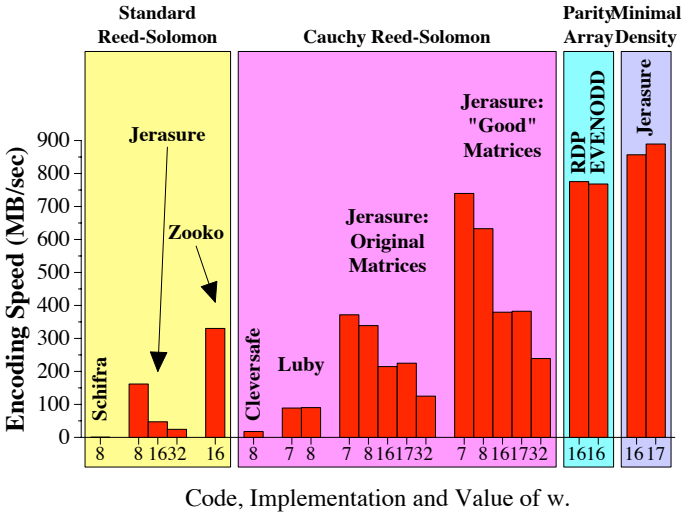

Figure 5: Encoding performance when $k = 14$ and $m = 2$.

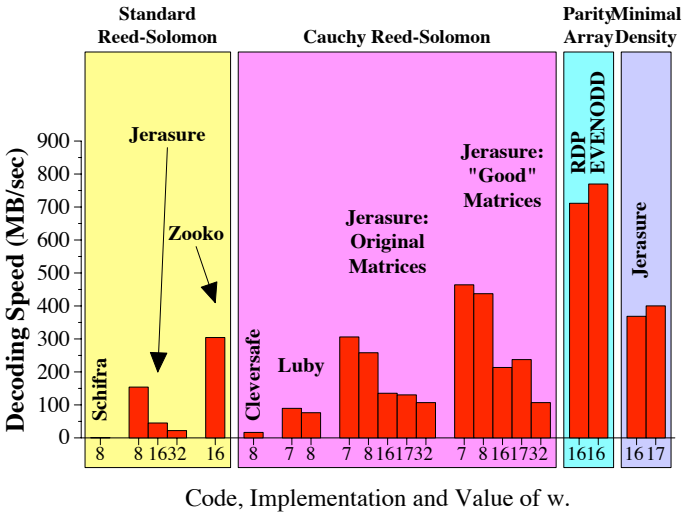

Figure 6: Decoding performance when $k = 14$ and $m = 2$.

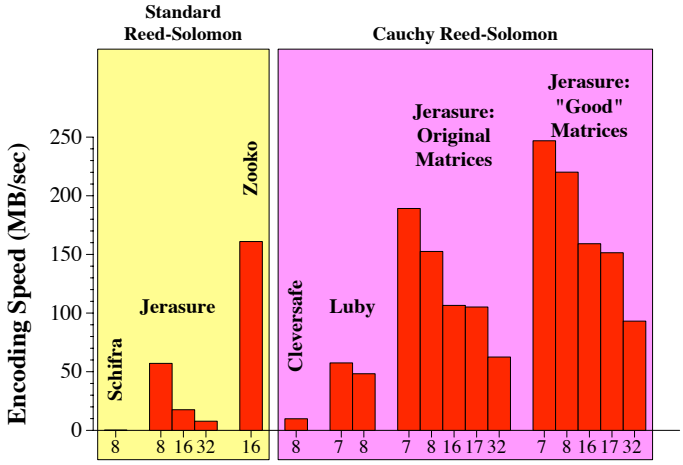

Figure 7: Encoding performance when $k = 12$ and $m = 4$.

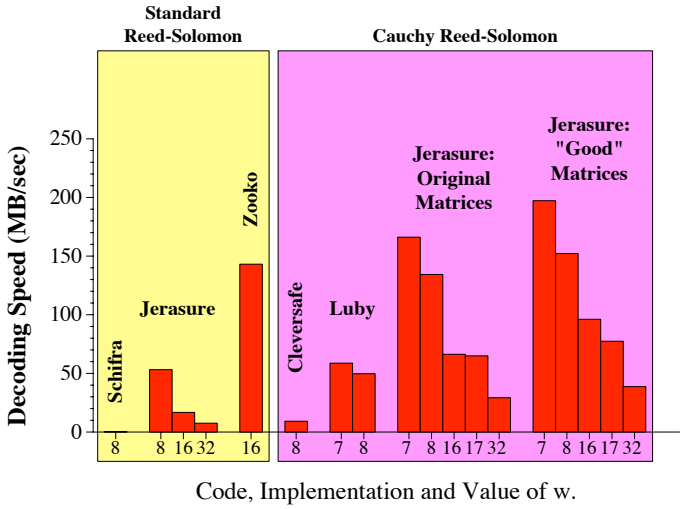

Figure 8: Decoding performance when $k = 12$ and $m = 4$.

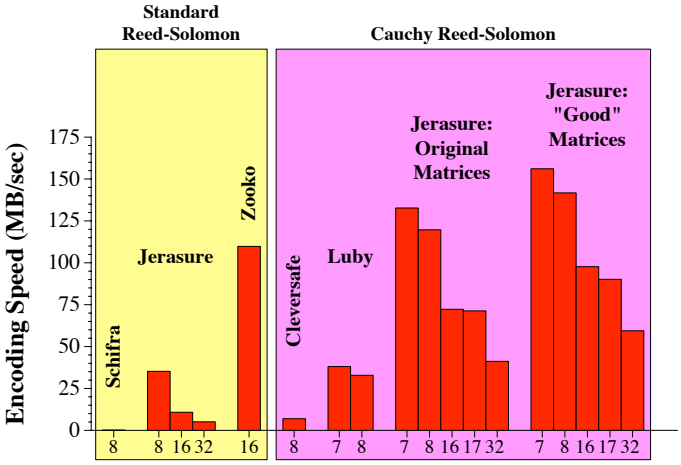

Figure 9: Encoding performance when $k = 10$ and $m = 6$.

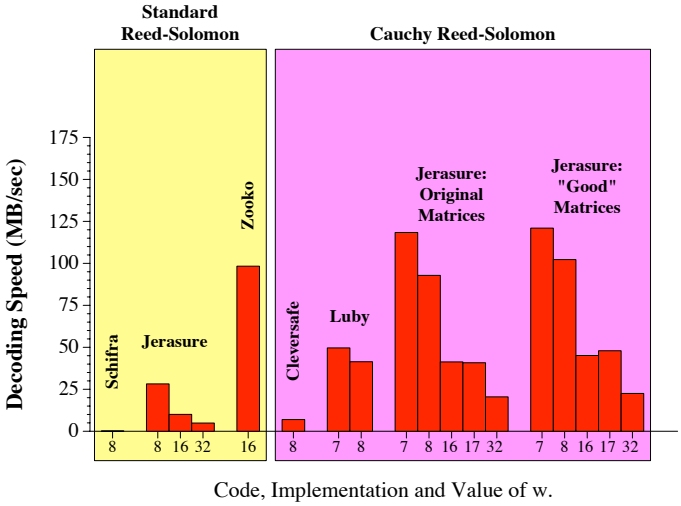

Figure 10: Decoding performance when $k = 10$ and $m = 6$.