Storage Architecture with Integrity, Redundancy and Encryption

Henning Klein
Fujitsu Siemens Computers GmbH
Buergermeister-Ulrich-Strasse 100
86199 Augsburg, Germany
Henning.Klein@fujitsu-siemens.com

Jörg Keller
FernUniversität in Hagen
Dept. of Mathematics and Computer Science
58084 Hagen, Germany
Joerg.Keller@fernuni-hagen.de

Abstract

We propose a storage system that treats confidentiality, integrity and availability of data in a unified manner. Extending RAID6, it allows for failures of multiple disks, encrypts data on disk, and stores checksums to detect faulty data without disks failing, which occurs e.g. in solid state disks due to wear out of cells. By handling encryption and integrity check together, the probability of undetected faulty data is reduced further. We provide an implementation, i.e. a driver, which encapsulates all these features and uses parallel algorithms exploiting multicore processor performance to match the bandwidth available from multiple disks. We present performance figures of our experiments.

1. Introduction

Valuable data stored in a computer system must be protected following the goals of confidentiality, integrity and availability. To achieve the first goal, data is encrypted on disk. To achieve integrity, checksums based on hash functions are computed and stored. To achieve availability, redundant storage is used to overcome disk failure. Typically, these measures are applied separately and on different levels. For example, integrity is often checked on file system level, while redundancy is often encapsulated in a RAID system. Integrity checks are of growing importance because with rising disk sizes and the introduction of solid state disks the probability of faulty blocks without a disk failure increases considerably due to cell wear-out.

We propose a software solution able to handle these three features together. The software is encapsulated in a driver, so that it can be incorporated into a system without distracting users. For fault tolerance, we build on a common RAID system. Yet we extend this RAID system by integrity checksums, with the advantage that those checksums can be incorporated into RAID redundancy data. Thus integrity check does not incur storage overhead. This in turn also leads to better alignment to RAID block sizes which delivers better performance than systems storing the parity separately. Data is encrypted symmetrically on block level by AES, where we build on an open-source AES implementation by Gladman [1]. By encrypting on block level, violation of integrity in one bit will spread over many bits during decryption. Thus, the probability of undetected faulty disk blocks is further decreased.

Doing all this in software requires lots of computations, so that the microprocessor might become the bottleneck compared to the aggregate bandwidth of multiple fast disks in a RAID system. Therefore, the driver uses parallel algorithms that exploit the current multicore processor architecture. We present performance figures from a prototype implementation that indicate that performance is competitive and scales well.

The remainder of this paper is structured as follows. Section 2 describes related work. In Section 3 the proposed technique and algorithm are introduced. Section 4 describes general implementation aspects. Section 5 validates the performance and error correction capability of a prototype implementation. In Section 6 we give a conclusion and an outlook on future work.

2. Related Work

RAID (Redundant Array of Inexpensive Disks) is a common technique used to increase performance, reliability or both. It has been introduced in [2]. Among the variety of RAID systems a disk array tolerating the failure of two disk losses, called RAID 6 as described in [3] has been investigated, which is able to tolerate two disk losses with two additional disks using Reed Solomon codes [4]. The main problem with this code are the Galois Field multiplications that don't perform well on standard PC processors. This can be avoided using Cauchy Reed Solomon Codes, that have been introduced in [5]. This algorithm can be applied using XOR operations only. This technique has been optimized to need fewer operations by changing the encoding matrices [6], [7]. There have also been investigations accelerating the encoding process by hardware with a FPGA-Coprocessor [8]. More specific codes have been developed like EVENODD [9], Blaum-Roth [10] or Liberation [11] amongst others. Depending on the number of data and parity disks and the bits per data unit (word size) one of the numerous algorithms is applicable and performs better or

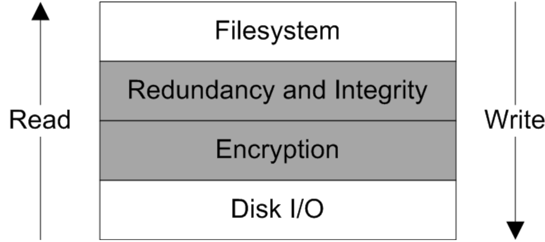

Figure 1. Architecture layers.

RAID	Disk 1	Disk 2	Disk 3	Disk 4	Disk 5
Stripe 0	D_{00}	D_{01}	D_{02}	P_{01}	Q_{01}
Stripe 1	D_{10}	D_{11}	D_{12}	R_{01}	S_{01}
Stripe 2	D_{20}	D_{21}	P_{23}	Q_{23}	D_{22}
Stripe 3	D_{30}	D_{31}	R_{23}	S_{23}	D_{32}

Figure 2. Position of parity and data blocks.

worse than others. In [12] an optimized RAID-6 implementation is presented that avoids the usage of lookup tables for multiplications to exploit wide data paths of modern personal computers. The present paper uses the same codes and RAID layout as introduced in [13]. We extend this solution by enhancing the code generation algorithm using an optimized lookup table for the number and size of parity bytes used. We provide a rough comparison to the next best applicable general algorithm as evaluated in [14]. Furthermore we combine the proposal from [13] with encryption to achieve confidentiality besides integrity and reliability, and use this combination to improve fault-detection capabilities as well.

The advanced encryption standard (AES), proposed by [15] and standardized by NIST [16], is currently the standard in encryption technology and the successor of DES. AES single core encryption has been implemented in various scenarios, for example in the filesystem ZFS [17] or as a partition encryption in the operating system Windows Vista [18]. Many hardware assisted speedup architectures like in [19] have been proposed. We use the AES block encryption to achieve confidentiality *and* to increase error correction capabilities. We employ a widely-used implementation of Gladman [1] to measure performance of the parallelized and combined technique.

The file system ZFS with features similar to the one presented in this paper has been developed by SUN [17]. Their implementation uses separated layers for encryption, integrity checking and fault tolerance. Our solution is filesystem independent and needs no extra storage space to save checksum values. It combines all computationally intensive operations to allow effective parallelization with little overhead.

3. General concept

The proposed technique uses a two-layered architecture. The layers are combined before computing on multiple processors to reduce overhead. If data is written to the disk, for example, redundancy and integrity algorithms are added before encrypting user and parity data, see Fig. 1. The data used for integrity checking is secured that way, too.

The proposed technique is based on Reed-Solomon correction codes as in a RAID 6 system. Like the well known RAID it uses two extra hard disk drives for storing redundant data (parity) of k data disk drives and is able to recover data in the case of the loss of up to two hard disks. Just like in RAID 6 systems the content of each disk drive is divided into equally sized blocks. One row of blocks across all disks in an array having the same offset is called a stripe. Two blocks of one stripe contain parity information instead of data. The parity information is not stored on two specific disks. It is spread across all disks, changing the position every two stripes, see grey blocks in Fig. 2. Instead of calculating two parity values like in RAID-6 the proposed system uses four values P, Q, R and S. The parity values are computed using the data across two stripes. Fig. 2 shows an example of the algorithm for an array of $k+2=5$ disks.

The algorithm for parity generation is based on the Galois Field GF(2^8) with generators g^n that are multiplied with data values of the disks:

$$\begin{aligned}
P &= D_{00} + D_{01} + ... + D_{10} + D_{11} + ... + D_{1k} \\
Q &= D_{00} \cdot g^0 + D_{01} \cdot g^1 + ... + D_{0k} \cdot g^{k-1} \\
&\quad + D_{10} \cdot g^k + D_{11} \cdot g^{k+1} + ... + D_{1k} \cdot g^{2k-1} \\
R &= D_{00} \cdot g^0 + D_{01} \cdot g^2 + ... + D_{0k} \cdot g^{2(k-1)} \\
&\quad + D_{10} \cdot g^{2k} + D_{11} \cdot g^{2(k+1)} + ... + D_{1k} \cdot g^{2(2k-1)} \\
S &= D_{00} \cdot g^0 + D_{01} \cdot g^3 + ... + D_{0k} \cdot g^{3(k-1)} \\
&\quad + D_{10} \cdot g^{3k} + D_{11} \cdot g^{3(k+1)} + ... + D_{1k} \cdot g^{3(2k-1)}
\end{aligned}$$

This enables the reconstruction of four data blocks within a set of two stripes, if the blocks can be identified as faulty. If disks are defect the missing blocks are known and can therefore be recovered. The price we pay for this advantage is that the maximum number k of data disks is reduced from 256 (RAID6) to 43, which we see still as sufficient for the surroundings we have in mind.

Integrity checks are being done by recalculating parity values from disk and comparing them to the values stored on disk. In principle integrity checks could as well be done with RAID-6 implementations but it would be limited to a maximum probability of $2^{-16} = 1.5 \cdot 10^{-5}$ if all disks are working properly and $2^{-8} = 3.9 \cdot 10^{-3}$ if one disk in the array fails The probabilities come from the number of parity bytes used (2 and 1 resp.). In the latter case errors can only be detected if the position of the faulty block is known. In our proposal more parity values are used and therefore the

capability of successful error detections is higher, see next chapter.

The encryption used in this scenario is 128 bit AES that encrypts 16 byte blocks. For performance measurement an implementation of [1] has been used.

To achieve competitive performance, the driver that implements the above features is multi-threaded, so that multicore performance can be exploited. Different threads concurrently process disk block data which is available in large numbers as a system typically has more disks than cores.

4. Implementation concepts

Our implementation focusses on performance. There are two main aspects that speed up the system: Using multiple cores or CPUs and creating an adjusted algorithm for parity encoding and decoding. In the solution proposed by [13] four parity values are used with the size of eight bits each, which sums up to 32 bit. To calculate these values the same data byte has to be multiplied with different values g^n. Therefore we use a lookup table which consists of 32 bit entries holding a precomputed result for the multiplication of the data byte with the corresponding g^ns for each disk. For the parity value P, which is computed without multiplications, we simply insert the corresponding data value. The resulting two dimensional array provides a table for each possible multiplication set and can be addressed by the data value and the position in the equation. A 32-bit value containing all of the four parity values can then computed by looking up $2k$ values and summing them up by $2k-1$ XOR operations which are 32 bits wide.

$$\text{table}[m][D_x] = \begin{pmatrix} P_m \\ Q_m \\ R_m \\ S_m \end{pmatrix} = \begin{pmatrix} D_x \\ D_x \cdot g^m \\ D_x \cdot g^{2m} \\ D_x \cdot g^{3m} \end{pmatrix}$$

$$\begin{pmatrix} P \\ Q \\ R \\ S \end{pmatrix} = \bigoplus_{m=0}^{2k-1} \begin{pmatrix} P_m \\ Q_m \\ R_m \\ S_m \end{pmatrix}$$

During this parity encoding the processor will load parts of the table into its cache to avoid longer lasting RAM transfers. For growing k, the data blocks together with tables get too large ($2^{11} \cdot k$ bytes) for current processor caches and so the encoding rate drops with the number of disks (see next chapter for performance measurements). Therefore we reschedule the order of accesses such that the algorithm uses a smaller set of tables within a certain time range. This is achieved by buffering intermediate results during parity computations and performing lookups for the same multiplication factor for all data bytes of one disk block. Fig. 3 depicts this process in detail.

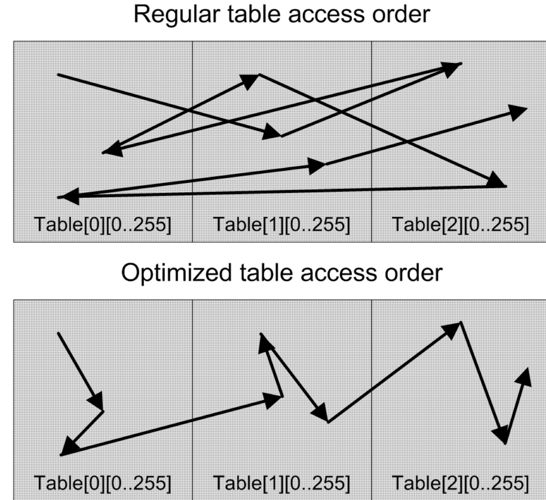

Figure 3. Optimizing lookup table access

The performance of our proposal can easily be increased by parallelizing the workload on different processors. We use block level parallelism to speed up computations under heavy load. By implementing encryption and parity generation within one system the overhead can be reduced and the workload can easier be balanced to all of the processors than in multi layered architectures.

A side effect of our combination of integrity check and encryption is to decrease the probability of uncorrectable defect disk blocks by including a block encryption algorithm. Depending on the underlying storage system it is not mandatory that occurrences of data errors are reported correctly [20]. In this case the algorithm has to be able to recognize errors, find their location and correct them. To achieve this the position of erroneous data has do be guessed by enumerating all possible combinations. Then the decoding algorithm has to be applied to reconstruct data. Finally the correction has to be verified. In our solution we use four parity values and try to correct up to three errors with unknown position by the following method: We need one parity byte to correct a possibly corrupted data byte. After correcting the data bytes the parity values will be recalculated and compared to the ones that have not been used for the correction process. If the values match a possible solution was found. If more failures occur, fewer parity values can be used to verify a proper correction and the probability for multiple solutions that are possibly correct increases. Using the block level encryption AES, a single bit error will alter the data of a whole encrypted block of 16 bytes. Knowing this, the algorithm can be changed to compare possible error positions 15 more times as the error will affect the whole encrypted block. Fig. 4 illustrates an example of connected AES blocks within two stripes on four disks. Up to three erroneous AES blocks can be corrected.

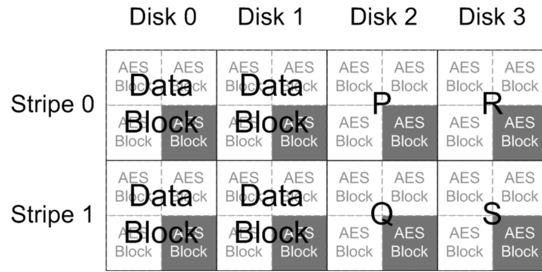

Figure 4. Connected AES blocks tolerating up to three errors

5. Performance and capabilities

The proposed technique was implemented in order to run experiments about error detection and correction capabilities. The implementation of the whole feature stack demonstrates competitive performance and scalability when using multiple processors or cores. A consumer system (Fujitsu Siemens Computers Amilo Li3740) with a 2.33 GHz Intel quadcore processor has been used to measure the data throughput when encrypting data and calculating parity values. We present performance and capability results for two setups. First we focus on the speed of the combined redundancy and integrity solution. Secondly we present results of the system after adding encryption. Our main focus is the performance of the parity generation which we also use for checking integrity.

Figure 5 depicts the performance of our specific solution compared to more general attempts. All techniques were run using one processor core with the same parameters: 8 bit word size, 4 parity values and 12 data disks. For comparison we use the GNU LGPL C Library Jerasure [21], Version 1.2 and ran it using SUSE Linux 11.1. Our measurements show that the Vandermonde and Cauchy Reed Solomon Matrix implementation ran significantly better than a solution with a single lookup table to speed up Galois Field multiplications. The fastest alternative is the XOR based "good" Cauchy Reed Solomon Algorithm, that has been improved by reducing the number of XOR operations compared to the original method [5]. Our solution with a 32 bit lookup table performed more than five times faster on the same machine than the next best implementation. A recent paper [14] reports improved results for tests utilizing this library, probably in a more optimized binary. But still our results are better for the specific scenario on similar hardware.

Figure 6 shows the multithreaded performance results for the main scenario, when all disks in the RAID array are healthy. The parity is generated twice: before writing and after reading data in order to create redundancy data and check integrity. The figure shows the simple lookup table technique using one table to speed up multiplications on a

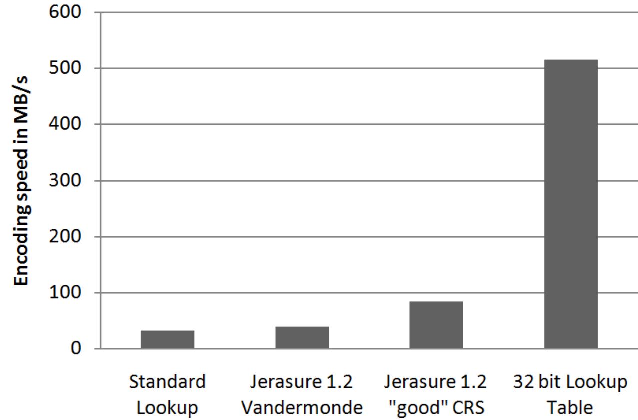

Figure 5. Performance of general solutions

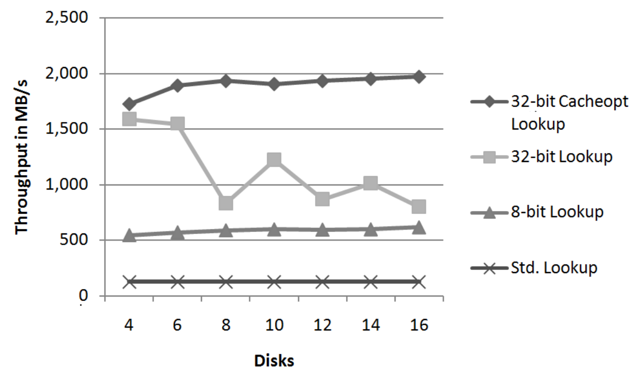

Figure 6. Comparison of implemented techniques on four Processors

Galois Field. The more advanced 8-bit Algorithm uses a two dimensional eight bit lookup table, that holds precomputed multiplications for terms of the type $D_x \cdot g^y$. If parity values are computed for four data disks, both of the 32 bit lookup table implementations showed good results beyond 1,5 GB/s as the size of the lookup table is still quite small. With an increasing number of disks the size of the table grows and the cache optimized solution performs clearly better, reaching almost 2 GB/s.

In Figure 7 we present the speedup between single and quad core execution. It underlines the importance of a cache-optimized implementation, which almost quadruples the speed.

Figure 8 shows the performance after one or two disks failed. There is still room for improvements as we did not optimize the order of table accesses. However, already our single core performance is comparable to the best solutions evaluated in [14].

The system is able to perform integrity checks as long as no more than one disk fails. If a second disk fails data can be recovered but not checked. Several billion test rounds

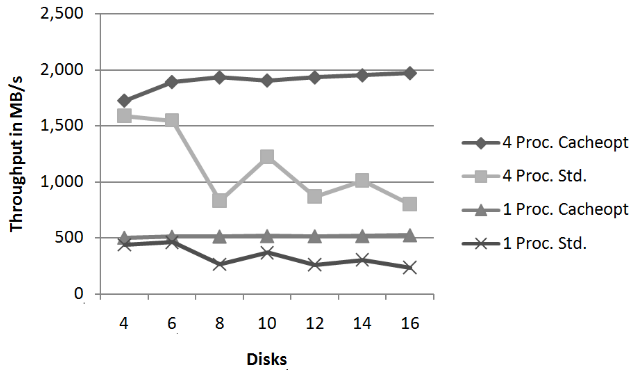

Figure 7. Comparison between single and four core execution

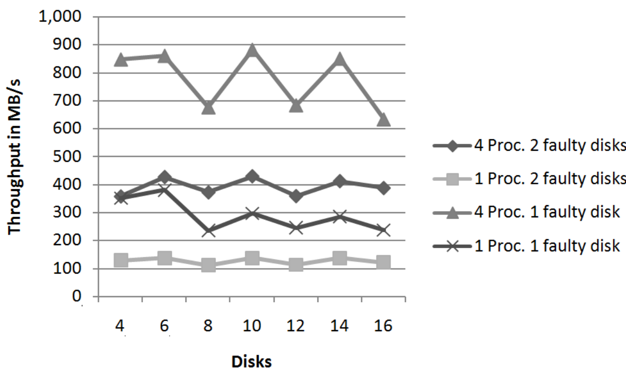

Figure 8. Data recovery performance after disk losses

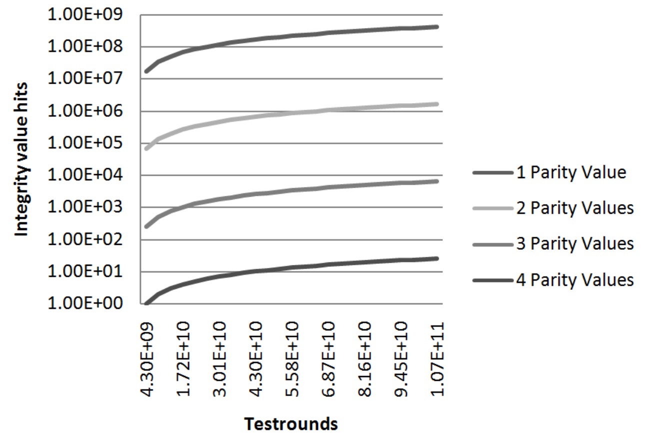

Figure 9. Error detection capability.

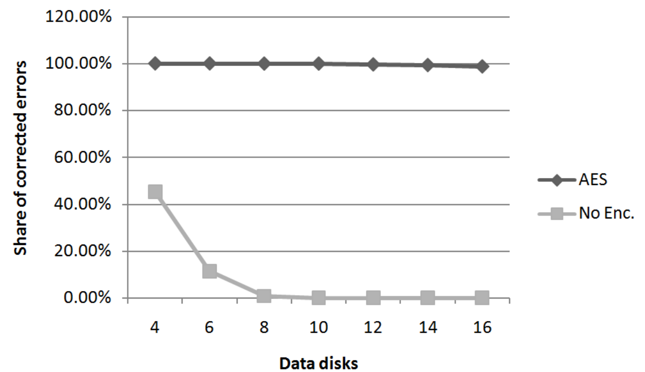

Figure 10. Error Correction Capability

were run on random data simulating four disks and random parity values. Correct parity values were computed and compared to the randomly generated values. Figure 9 shows the capabilities in both cases. If no disk fails, matching combinations of random data and the corresponding correct random parity values are generated once in $4.3 \cdot 10^9$ test rounds. If one disk fails, $6.5 \cdot 10^4$ only test rounds were necessary to create a randomly matching combination of data and parity. This capability could be increased if six parity values are used within three stripes on the cost of higher computation times.

Using a combination of fault tolerance and encryption has advantages for the recovery of faulty blocks, too. The proposed system is able to detect and recover three errors in two stripes, as long as no disk failed. With a rising number of disks in the array, the capability of successful data recovery decreases. The proposed technique uses AES with 16 byte blocks which is applied by decrypting immediately before error detection and correction. If a single byte has changed within a 16 byte block, the remaining 15 bytes will most definitely change after decryption, too. Therefore the positions of any computed possible solution for three simultaneous errors can be checked 15 times when correcting the rest of the block. However, if errors cannot be corrected, a single byte error would end up in 16 corrupted bytes after decryption. Another effect is that the position of three errors must not change within 16 bytes. Figure 10 shows the improvement of error correction capabilities before and after adding encryption and the awareness in the correction algorithm. 8192 triple errors were injected in random generated data. Using encryption the system was able to correct almost all errors. Single and double errors within two stripes have always been corrected, regardless if encryption was used or not.

Finally, Fig. 11 shows the overall performance of an architecture with integrity, redundancy and encryption utilizing one and four cores. The speed can almost be quadrupled on quad core systems comparing to a single core execution. The figure shows that the transfer rate increases with a larger number of disks. The reason for this is the changing ratio of user and parity data. We measure the throughput of processed "user" data per second, but the parity is encrypted as well to avoid the reconstruction of user data. Using $k = 4$ data disks the share of encrypted parity is $2/(k+2) = 33\%$

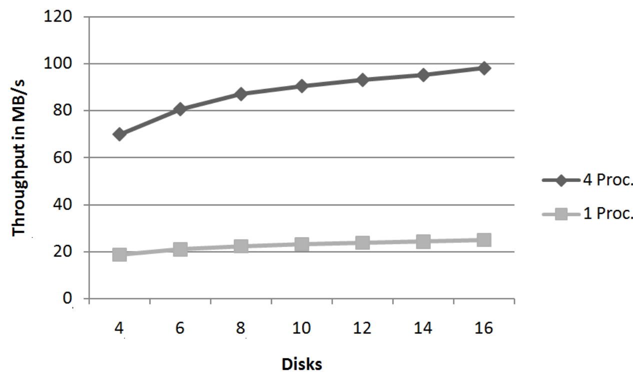

Figure 11. Combined features performance

of the total data. If $k = 16$ data disks are used this share decreases to 11%.

6. Conclusions

We introduced a technique offering integrity, security and redundancy for storage systems. This combination allows a design using little extra storage capacity to tolerate two disk failures and integrity checking with a high error detection capability. Additionally, a flexible error correction mechanism can be achieved. Our implementation on modern multicore processors showed competitive performance even though completely implemented in software. The proposed scheme can be used to design a reliable and secure environment using common PC hardware.

References

[1] B. Gladman, "AES and combined encryption/authentication modes," http://fp.gladman.plus.com/AES/.

[2] D. A. Patterson, G. Gibson, and R. H. Katz, "A case for redundant arrays of inexpensive disks (RAID)," in *SIGMOD '88: Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, 1988, pp. 109–116.

[3] J. S. Plank, "A tutorial on Reed-Solomon coding for fault-tolerance in RAID-like systems," *Softw. Pract. Exper.*, vol. 27, no. 9, pp. 995–1012, 1997.

[4] I. S. Reed and G. Solomon, "Polynomial codes over certain finite fields," *Journal of the Society for Industrial and Applied Mathematics*, vol. 8, no. 2, pp. 300–304, 1960.

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

[6] J. S. Plank, "A new minimum density RAID-6 code with a word size of eight," in *Proc. 7th IEEE International Symposium on Network Computing and Applications (NCA '08)*, Cambridge, MA, Jul. 2008, pp. 85–92.

[7] J. S. Plank and L. Xu, "Optimizing Cauchy Reed-Solomon codes for fault-tolerant network storage applications," in *Proc. 5th IEEE International Symposium on Network Computing and Applications (NCA '06)*, 2006, pp. 173–180.

[8] V. Hampel, P. Sobe, and E. Maehle, "Experiences with a FPGA-based Reed/Solomon-encoding coprocessor," *Microprocess. Microsyst.*, vol. 32, no. 5-6, pp. 313–320, 2008.

[9] M. Blaum, J. Brady, J. Bruck, and J. Menon, "EVENODD: An efficient scheme for tolerating double disk failures in raid architectures," *IEEE Transactions on Computers*, vol. 44, no. 2, pp. 192–202, 1995.

[10] R. Blaum and R. M. Roth, "On lowest density MDS codes," *IEEE Transactions on Information Theory*, vol. 45, no. 1, pp. 46–59, 1999.

[11] J. S. Plank, "The RAID-6 Liberation codes," in *Proc. 6th Usenix Conference on File and Storage Technologies (FAST-2008)*, Feb. 2008, pp. 97–110.

[12] H. Anvin, "The mathematics of RAID-6," 2009, www.kernel.org/pub/linux/kernel/people/hpa/raid6.pdf.

[13] H. Klein and J. Keller, "RAID architecture with correction of corrupted data in faulty disk blocks," in *Proc. 6th ARCS Workshop on Dependability and Fault-Tolerance*, Delft, NL, Mar. 2009.

[14] J. S. Plank, J. Luo, C. D. Schuman, L. Xu, and Z. Wilcox-O'Hearn, "A performance evaluation and examination of open-source erasure coding libraries for storage," in *Proc. 7th Usenix Conference on File and Storage Technologies (FAST-2009)*, Feb. 2009, pp. 253–266.

[15] J. Daemen and V. Rijmen, "AES proposal: Rijndael," http://www.esat.kuleuven.ac.be/rijmen/rijndael/.

[16] National Institute of Standards and Technology, "Specification for the advanced encryption standard (AES)," Nov. 2001, FIPS PUBS 197.

[17] Sun Microsystems, "Sun on-disk specification," http://opensolaris.org/os/community/zfs/docs/ondiskformat-0822.pdf.

[18] Microsoft, "Bitlocker drive encryption," http://technet.microsoft.com/en-us/windows/aa905065.aspx.

[19] C. Caltagirone and K. Anantha, "High throughput, parallelized 128-bit AES encryption in a resource-limited FPGA," in *Proc. 15th ACM Symposium on Parallel Algorithms and Architectures*, 2003, pp. 240–241.

[20] J. Bonwick and B. Moore, "ZFS — the last word in file systems," http://opensolaris.org/os/community/zfs/docs/zfs_last.pdf.

[21] J. S. Plank, S. Simmerman, and C. D. Schuman, "Jerasure: A library in C/C++ facilitating erasure coding for storage applications - Version 1.2," University of Tennessee, Tech. Rep. CS-08-627, Aug. 2008.