RAIDq: A software-friendly, multiple-parity RAID

Ming-Shing Chen[†], Bo-Yin Yang[‡], and Chen-Mou Cheng[†‡]
[†]*Department of Electrical Engineering, National Taiwan University, Taiwan*
[‡]*Research Center for Information Technology Innovation, Academia Sinica, Taiwan*

Abstract

As disk manufacturers compete to build ever larger and cheaper disks, the possibility of RAID failures becomes more significant for larger and larger disk arrays, creating opportunities for products beyond RAID 6. In this paper, we present the design and implementation of RAIDq, a software-friendly, multiple-parity RAID. RAIDq uses a linear code with efficient encoding and decoding algorithms and addresses a wide range of general cases of RAID that are of practical interest. However, RAIDq does have a limit on how many data disks it can support, which we will analyze in this paper. A second benefit of RAIDq is that it includes existing RAID 5 and 6 as special cases and hence is 100% backward compatible. This allows RAIDq to reuse the efficient coding algorithms and implementations of RAID 5 and 6. Last but not least, RAIDq is optimized for software implementation, as its encoding only involves simple `XOR` and multiplication by several fixed elements in a finite field. Thanks to the popularity of RAID 6, such operations have been highly optimized on modern processors, of which RAIDq can take advantage, as corroborated by our experiment results.

1 Introduction

Since its inception in the early eighties, RAID has been a mainstream technology to increase the performance and reliability of computer storage systems at a cost of using redundant disks for storing checksum data. For many applications, RAID 5 and 6 are the preferred standards because they allow efficient encoding and decoding, thus providing a good trade-off among performance, reliability, and cost.

Nevertheless, the RAID technology does seem to start showing some wear over the years. As disk manufacturers compete to build ever larger and cheaper disks, the possibility of RAID failures becomes more significant for RAID 5 and 6 products in the mainstream market [4]. Although there are several alternative methods of extending beyond RAID 6, use of Reed-Solomon or similar codes still seems the most attractive because they can generate an (almost) arbitrary amount of checksum data for a wide range of disk-array sizes and configurations [8]. The downside of using these codes is that they often involve general finite-field operations in encoding and decoding, which tend to be much more expensive than those for RAID 5 and 6. This is possibly a major reason why there has not been wide industrial adoption of RAID technologies beyond RAID 6, as cost is a major concern especially for entry-level products.

Why have there not been efficient constructions like RAID 5 and 6 but with more redundant disks? As pointed out by Leventhal in 2010, RAID 5 and 6 probably represent two special cases for which there are highly efficient specialized encoding and decoding algorithms [4]. He concluded: "Not only is there a need for triple-parity RAID, but there's also a need for efficient algorithms that truly address the general case of RAID with an arbitrary number of parity devices."

In this paper, we present our answer to Leventhal's call: a software-friendly, multiple-parity RAID called *RAIDq*. The construction of RAIDq allows efficient encoding and decoding algorithms, and it *somewhat* addresses the general case of RAID with an arbitrary amount of checksum data *up to certain limits*. We can easily compute these limits, and in most cases, they are quite generous for practical RAID systems. That is, when we try to scale up and build a large RAID system, it is more likely that we hit some system or hardware limits such as bus bandwidth before we hit the coding limits.

Our construction was inspired by Plank's influential paper [5]. For ease of exposition, we shall henceforth use the following terminology.

Definition 1 *A* Plank's code *is a linear* $[n+m,n]$ *code over* $\mathbb{F}_q$ *whose generator matrix A is of the (systematic) form* $A = \begin{bmatrix} I_n & F \end{bmatrix}$. *Here F is an* $n \times m$ *Vandermonde* *matrix* $\mathrm{Van}(m;\alpha_0,\ldots,\alpha_{n-1}) := [\alpha_i^j]_{i=0,\ldots,n-1,j=0,\ldots,m-1}$, *where α_i's are distinct nonzero elements,* not necessarily generators*, of $\mathbb{F}_q$, and I_n the $n\times n$ identity matrix. This will be termed the* Plank's code generated by $[\alpha_0,\ldots,\alpha_{n-1}]$. *Usually we set $\alpha_0 := 1$. When $\alpha_i := \alpha^i$ for some fixed α, we call it the* Plank's code generated by *α (of length $n+m$ and rank n over $\mathbb{F}_q$).*

With $\alpha_i := \alpha^i$ for a generator $\alpha \in \mathbb{F}_q$, Plank's code would allow very efficient encoding, erasure decoding, and even error correction, as implemented and discussed by Anvin [1]. For $m = 1$ or 2, they are *exactly* the checksums used in RAID 5 and 6. However, Plank's codes, even those generated by one element, turn out to be *neither identical nor isomorphic* to Reed-Solomon codes, and the highly efficient constructions of Anvin [1] are maximum distance separable (MDS) over $\mathbb{F}_{256}$ only for $m = 1$, 2, or 3. In fact, Plank himself discovered this and corrected this unfortunate error in a subsequent note [6].

Nevertheless, RAIDq still uses Plank's codes because we believe that they are actually suitable for building general RAID systems beyond RAID 6. As we shall see, one can build a reasonably large RAID system by carefully choosing appropriate α_i's. Furthermore, although finite-field operations are involved in encoding and decoding, these operations can be significantly accelerated by various SIMD (single-instruction, multiple-data) instructions commonly found on modern processors. For example, `PSHUFB` (Packed-Shuffle-Byte) is a 16-entry table look-up instruction common to latest x86 processors, with which we will discuss how to accelerate multiplication in small binary fields in Section 3. Lastly, some of the latest Intel processors with built-in I/OAT DMA engine have special circuitry to accelerate the $\mathbb{F}_{256}$ operations in RAID 6. Such special circuitry can certainly be used to accelerate the encoding and decoding of a specialized class of Plank's codes.

2 Plank's codes for RAID

2.1 Why Plank's RAID >7 does not work

In the rest of this paper, we will call triple-parity RAID as RAID 7, quadruple-parity RAID as RAID 8, etc. When we examine Anvin's construction of RAID 7 based on Plank's code generated by α of length $n+3$ over $\mathbb{F}_{256}$, we see that it works fine because the code is MDS. This is because all square submatrices of $\mathrm{Van}(3;\alpha^i,\alpha^j,\alpha^k)$ are of full rank, i.e., whose determinants are never zero as long as $0 \le i < j < k < 255$.

To extend this construction to RAID 8 over $\mathbb{F}_{256}$, *these determinants must be nonzero*: $(I) := \begin{vmatrix} 1 & \alpha^{3i} \\ 1 & \alpha^{3j} \end{vmatrix}$, $(II) := \begin{vmatrix} 1 & \alpha^{i} & \alpha^{3i} \\ 1 & \alpha^{j} & \alpha^{3j} \\ 1 & \alpha^{k} & \alpha^{3k} \end{vmatrix}$, and $(III) := \begin{vmatrix} 1 & \alpha^{2i} & \alpha^{3i} \\ 1 & \alpha^{2j} & \alpha^{3j} \\ 1 & \alpha^{2k} & \alpha^{3k} \end{vmatrix}$. $(I) \neq 0$ unless $i \equiv j \pmod{85}$; (II) is equal to $(\alpha^i-\alpha^j)(\alpha^i-\alpha^k)(\alpha^j-\alpha^k)(\alpha^i+\alpha^j+\alpha^k)$, and (III) can be written as $\alpha^{3(i+j+k)}(\alpha^{-i}-\alpha^{-j})(\alpha^{-i}-\alpha^{-k})(\alpha^{-j}-\alpha^{-k})(\alpha^{-i}+\alpha^{-j}+\alpha^{-k})$. Since $\alpha \mapsto \alpha^{-1}$ is an automorphism in $\mathbb{F}_{256}$, we know (II) and (III) vanish together or not at all. Using the same α as in most RAID 6 implementations, we can only go as high as 21. Choosing α as some other generators can lead to the determinants being nonzero for distinct $i, j, k < 28$, but no α gets us to 28 or higher.

Therefore, following the original narrative of Anvin [1], we can construct RAID 8 (quadruple-parity) over $\mathbb{F}_{256}$ up to 21+4 disks, or 27+4 if compatibility with existing RAID 6 is disregarded.

We note that the 21+4 limit with the standard α is not "unluckily small." Let us re-examine the regularity check above for the original quadruple-parity Plank's code. Without loss of generality, we can assume $i = 0$. If $1+\alpha^j+\alpha^k$ is essentially a random number in $\mathbb{F}_{256}$ as $j < k$ varies, one of the determinants (II) will become zero as we reach n data disks, where $\binom{n}{2} \approx q = 255$, or $n \approx 22$. The standard α is seen then to give us almost exactly the average result.

In fact, the regularity check for RAID 7 using a Plank's code based on $[1,a,b]$ would require $\begin{vmatrix} 1 & 1 & 1 \\ 1 & a^j & a^k \\ 1 & b^j & b^k \end{vmatrix} \neq 0$ for all j,k. If we again assume that a and b are unrelated, this would again be nearly random in $\mathbb{F}_{256}$ as j,k change, and we would not be able to use more data disks than $\gtrsim \sqrt{2q}$. It is only because the highly nonrandom set of generators $[1,\alpha,\alpha^2]$ was used that we obtain an MDS code here.

2.2 Constructing RAIDq 7 and 8

The common implementation of RAID 6 is over $\mathbb{F}_{256}$ with the representation $\mathbb{F}_{256} := \mathbb{F}_2[x]/(x^8+x^4+x^3+x^2+1)$ ("`0x11d` $\mathbb{F}_{256}$") and generator $\alpha := x$ ("`0x02`"). We use a notation of $[1,\alpha]$ as "checksum generators" for representing the two checksums in RAID 6. That is, the starting "1" represents the first `XOR` checksum $s_0 = \sum_i d_i$, and "α" represents the second checksum $s_1 = \sum_i d_i\alpha^i$, for $i = 0,\ldots,n-1$.

In constructing RAIDq, we seek more efficient "checksum generators." Our selection criteria include: (1) compatible with RAID 6 checksums and easy to accelerate with RAID 6 hardware; (2) allowing maximal number of data disks while being MDS; and (3) minimizing the computational cost by restricting to simple `XOR` and (preferably) multiplication by low powers of α. We note that due to the nature of RAID applications, we set higher priorities on encoding than erasure decoding, followed by error correcting, when it comes to minimizing computational cost.

2.2.1 Improvement of RAIDq 7 over RAID 7

If there is specialized circuitry to compute the second RAID 6 checksum $\sum_i d_i\alpha^i$ (just this particular form, not arbitrary power series), then instead of using $[1,\alpha,\alpha^2]$ as the checksum generators for RAID 7, our recommendation is to use $[1,\alpha,\alpha^{\frac{1}{2}}]$ for a better utilization of the specialized RAID 6 accelerator. With the $\alpha^{\frac{1}{2}}$ as the third checksum generator, the checksum calculation can be divided into two passes of RAID 6 checksum computations:

$$\begin{aligned}\sum_{i=0,1,\cdots} d_i\alpha^{\frac{i}{2}} &= \sum_{i=0,2,4,\cdots} d_i\alpha^{\frac{i}{2}} + \alpha^{\frac{1}{2}}\left(\sum_{i=1,3,5,\cdots} d_i\alpha^{\frac{(i-1)}{2}}\right)\\ &= \sum_{i=0,1,2,\cdots} d_{2i}\alpha^{i} + \alpha^{\frac{1}{2}}\left(\sum_{i=0,1,2,\cdots} d_{2i+1}\alpha^{i}\right).\end{aligned}$$

2.2.2 RAIDq 8 based on specialized Plank's codes

We seek elements of the form $\alpha^{1/i}$ as checksum generators to construct RAIDq 8 that scales better than the 21+4 RAID 8 based on the $[1,\alpha,\alpha^2,\alpha^3]$ Plank code described in Section 2.1. $[1,\alpha,\alpha^{1/3},\alpha^{2/3}]$ is our first candidate, which increases the range of being MDS from 21 to 63 data disks with RAID 6 compatibility. If we take square roots and use $[1,\alpha^{1/2},\alpha^{1/6},\alpha^{1/3}]$ instead, we can have a higher utilization of existing RAID 6 hardware accelerators at a cost of losing compatibility.

There is only one (rather minor) technical detail: $\alpha^{1/3}$ is not in $\mathbb{F}_{256}$, so we have to work in an extension field that contains $\alpha^{1/3}$, the smallest being $\mathbb{F}_{2^{24}} := \mathbb{F}_{256}[Y]/(Y^3+\alpha)$. We choose this particular representation of $\mathbb{F}_{2^{24}}$ because here $\alpha^{1/3}$ and $\alpha^{2/3}$ have rather sparse polynomial representations: $\alpha^{1/3} := \alpha^{85}Y$ (`0x00d600`), and $\alpha^{2/3} := \alpha^{170}Y^2$ (`0xd70000`), making multiplication by them much cheaper.

2.2.3 RAIDq 8 based on general Plank's codes

Besides generating all checksum generators as α^i with only *one* α, we further investigate general choices of checksum generators in a Plank's code. Furthermore, we shall not restrict ourselves to $\mathbb{F}_{256}$, as we have seen above that working in an extension field might help improve error-control capabilities without sacrificing encoding/decoding performance. Indeed, we have found a quite efficient quadruple-parity RAID construction over $\mathbb{F}_{2^{16}} := \mathbb{F}_{256}[X]/(X^2+\alpha^3X+1)$ using $[1,\alpha,\alpha^{1/2},X]$ as the checksum generators. We note that three out of the four checksums are actually in $\mathbb{F}_{256}$, and the construction is compatible with Plank's RAID 7. Here $\mathbb{F}_{2^{16}}$ elements are represented as linear polynomials in $\mathbb{F}_{256}[X]$, and "X" is the indeterminate (`0x0100` in binary representation).

Table 1: Proposed candidates for RAIDq 8

Checksum generators	Max. n	Working field
$[1,\alpha,\alpha^{1/2},\alpha^{3/2}]$	21	$\mathbb{F}_{256}$
$[1,\alpha,\alpha^{1/3},\alpha^{2/3}]$	63	$\mathbb{F}_{2^{24}}$
$[1,\alpha,\alpha^{1/2},X]$	92	$\mathbb{F}_{2^{16}}$

Although this is not an MDS code, the maximal number of data disks for the $[1,\alpha,\alpha^{1/2},X]$ construction is 92, which we believe should be more than enough for many practical RAID applications. These proposed candidates of RAIDq 8 are summarized in Table 1.

2.3 Erasure decoding and error correction

In erasure decoding, in order to reuse the highly optimized encoder, we can replace the missing data symbols with 0 and encode them again to get a set of new checksums. The missing symbols can then be solved from the linear system composed of the checksum differences and part of the generating matrix corresponding to the positions of the missing symbols. For small RAID systems, it is possible to enumerate all erasure modes, and hence we can prepare all possible inverse matrices to avoid computing them on the fly in decoding.

Plank's codes also allow correction of errors in unknown positions up to half the number of parity disks, which is a supplementary yet important feature for many RAID applications. The error correction mode of Plank's code is very similar to that of Reed-Solomon codes. For the $[1,\alpha,\alpha^{1/2},X]$ Plank's code recommended for RAIDq 8, however, the error decoder of general Plank's code is *not* applicable, as the fourth checksum is no longer in the ascending form of α^i. In this case, we can *guess* one error position and then solve the other error via general Plank's checksums. The last checksum in the extension field can then be used to confirm the result.

3 Hardware acceleration of $\mathbb{F}_{2^n}$ arithmetic

We suggest to implement fast and high-throughput arithmetic in small binary fields using SIMD table look-up instructions, e.g., `PSHUFB` on Intel x86 processors. Although the mnemonic `PSHUFB` may indicate that it was designed for data movement, we can use this instruction for simultaneously looking up 16 values from a 16-byte table of 4-bit entries. This instruction is available in all new Intel and AMD processors, and a similar instruction exists in ARM processors with NEON extensions.

3.1 Scalar-vector multiplication in $\mathbb{F}_{256}$

The multiplication of a 16-wide vector in $\mathbb{F}_{256}$ by an arbitrary scalar can be accomplished with two PSHUFB operations [3]. More precisely, suppose we need to multiply a vector of elements $a^{(i)}$ by a scalar b in $\mathbb{F}_{256}$. We can cut each $a^{(i)}$ into two nybbles corresponding to higher- and lower-degree parts as follows:

$$\begin{aligned} a^{(i)} \times b &= \left(\left(a_7^{(i)} x^7 + \cdots + a_0^{(i)}\right) b(x)\right) \\ &= \left(a_7^{(i)} x^3 + \cdots + a_4^{(i)}\right)\left(x^4 b(x)\right) + \left(a_3^{(i)} x^3 + \cdots + a_0^{(i)}\right) b(x). \end{aligned}$$

Therefore, we can obtain the desired result by loading two pre-computed multiplication results of $x^4 b(x)$ and $b(x)$, followed by two PSHUFB's for table look-up and an XOR for combining results.

Only a few fixed scalar-vector multiplications are needed in the encoding of Plank's codes. We can therefore optimize for these specific scalars instead of using the general $\mathbb{F}_{256}$ multiplication. Let us start with the obvious one, namely, the multiplication with the α that is a simple polynomial x (0x02 in 0x11d $\mathbb{F}_{256}$).

$$\begin{aligned} c \times \alpha &= x \cdot (c_7 x^7 + \cdots + c_0) \bmod (x^8 + x^4 + x^3 + x^2 + 1) \\ &= c_7 x^8 + \left(c_6 x^7 + \cdots + c_0 x\right) \bmod (x^8 + \cdots) \\ &\to c_7\left(x^4 + x^3 + x^2 + 1\right) + \left(c_6 x^7 + \cdots + c_0 x\right). \end{aligned}$$

The reduction $c_7 x^8 \bmod (x^8 + x^4 + x^3 + x^2 + 1)$ is accomplished by conditional adding $x^4 + x^3 + x^2 + 1$. The standard way to compute this in C language is `result = ((c<<1)&0xff)^((c&0x80)?0x1d:0)`. One can also optimize with PSHUFB, yielding the complementary `((c&0x80)?0:0xe2)` in certain situations.

Finally, multiplication by other small powers of α between α^{-4} and α^4 can be efficiently implemented using one fewer PSHUFB as compared to multiplication by a generic element as follows:

$$\begin{aligned} c \times \alpha^3 = &\left(c_4 x^7 + \cdots + c_0 x^3\right) + \\ &\left(c_7 x^{10} + c_6 x^9 + c_5 x^8 \bmod (x^8 + x^4 + x^3 + x^2 + 1)\right). \end{aligned}$$

The modular part still requires one PSHUFB, but the other PSHUFB can be replaced by a logical shift (PSLLW or PSRLW). This results in higher throughput and lower latency, as well as lower register pressure. However, we note that the lack of byte-sized PSLLB or PSRLB makes things more awkward than expected.

Unlike the constant-time $\mathbb{F}_{256}$ multiplication on RAID 6 hardware, the performance of $\mathbb{F}_{256}$ arithmetics in software may change depending on the implementation. The benchmark results of some of the proposed SSSE3 implementations are summarized in Table 2.

Table 2: Cycle counts of $\mathbb{F}_{256}$ operations for 4 KB data on Intel Xeon E5430 @ 2.66 GHz

$\mathbb{F}_{256}$ operation	Cycle count
$\times\alpha$ (SSE2)	914
$\times\alpha$	725
$\times\alpha^3$	900
× arbitrary element	1187

3.2 Multiplication in $\mathbb{F}_{2^{16}}$ and $\mathbb{F}_{2^{24}}$

$\mathbb{F}_{2^{16}}$ and $\mathbb{F}_{2^{24}}$ are implemented as extension fields from 0x11d $\mathbb{F}_{256}$, which allows us to reuse some of the highly-optimized implementations of $\mathbb{F}_{256}$ arithmetics. Furthermore, a multiplication by an element in $\mathbb{F}_{256}$ would be preserved byte-wise in the extension fields.

For general multiplication in extension field, multiplication $a(X) \cdot b(X)$ of two arbitrary $\mathbb{F}_{2^{16}}$-elements are implemented with Karatsuba [2] as a polynomial multiplication over $\mathbb{F}_{256}$. Similarly, 3-way Karatsuba [2] is also applicable to arbitrary $\mathbb{F}_{2^{24}}$ multiplications.

For a hardware that enables us to do a long vector (block) checksum in DMA fashion over $\mathbb{F}_{256}$, i.e., high-speed, offloaded evaluation of $\sum d_j \alpha^j$ or even any $\sum d_j a_j$ for specified coefficients a_j, these can be applied toward arithmetic in $\mathbb{F}_{2^{16}}$ and $\mathbb{F}_{2^{24}}$ as well. For example, for data blocks D_i (considered as long vectors of $\mathbb{F}_{2^{16}}$), we can write down the checksum $\sum D_i X^i$ as

$$\begin{aligned} &D_0 + X D_1 + \left(1 + \alpha^3 X\right) D_2 + \left(\alpha^3 + (1 + \alpha^6) X\right) D_3 \\ &\quad + \left((1 + \alpha^6) + \alpha^9 X\right) D_4 + \left(\alpha^9 + (1 + \alpha^6 + \alpha^{12}) X\right) D_5 \\ &\quad + \left((1 + \alpha^6 + \alpha^{12}) + \alpha^{15} X\right) D_6 + \cdots. \end{aligned}$$

We can see clearly that it does have some regular structure, which leads to efficient use of any special hardware already in place to sum power series in α over $\mathbb{F}_{256}$ after we split each element of $\mathbb{F}_{2^{16}}$ into two halves .

Finally, we compare our multiplication techniques with those proposed by Plank, Greenan, and Miller [7]. In their proposal, a 16-bit element is divided into four nybbles, and a multiplication can be computed using eight table look-ups (two for each nybble). This technique is applicable to all $\mathbb{F}_{2^{16}}$ representations at a cost of storing huge pre-computed look-up tables. Specifically, it requires 8 × 16 bytes of storage for each $\mathbb{F}_{2^{16}}$ element (65536 × 128 bytes in total), which cannot possibly fit into the L1 cache of any of today's processors. Our tower field-based techniques, on the other hand, require much less fast memory, not to mention that they also involve less table-ups at run time.

Table 3: Erasure-coding throughput (MB/s) for $n = 16$

Code	m	Max. n	Encode	Decode
$[1,\alpha]$ SSE2 (Linux)	2	-	8762	-
$[1,\alpha]$	2	-	9051	6467
Reed-Solomon	2	-	3833	3517
$[1,\alpha,\alpha^2]$	3	-	5147	3276
$[1,\alpha,\alpha^{1/2}]$	3	-	4707	3542
Reed-Solomon	3	-	2977	2444
$[1,\alpha,\alpha^2,\alpha^3]$	4	21	3818	2478
$[1,\alpha,\alpha^{1/2},\alpha^{3/2}]$	4	21	3225	2338
$[1,\alpha,\alpha^{1/3},\alpha^{2/3}]$	4	63	2387	1290
$[1,\alpha,\alpha^{1/2},X]$	4	92	3186	1772
Reed-Solomon	4	-	2490	1921

Table 4: Error-correcting performance for $n = 16$

$[1,\alpha,\alpha^{1/2},X]$ RAIDq 8	Throughput (MB/s)
Encode	3187
Decode (0 errors)	2695
Decode (2 errors)	87

4 Experiments and concluding remarks

4.1 Experiment setup and results

We have implemented several RAID encoders and decoders, including a Reed-Solomon coder, in C language and Intel SIMD intrinsics. Some coders, such as the encoder of $[1,\alpha]$ and $[1,\alpha,\alpha^2]$, are further hand-optimized using assembly language. For erasure coding, the $[1,\alpha]$ SSE2 implementation is taken from the RAID 6 code in Linux kernel, which is implemented by Anvin [1] in assembly language without our SSSE3 optimizations. We leave its decoding performance blank because the decoding involves general $\mathbb{F}_{256}$ multiplications, and there is no natural SSE2 implementation except changing the data layout to a "bitsliced" form [3]. *All the other implementations have our SSSE3 optimizations.*

The experiments on erasure coding and error correcting are performed on an Intel Xeon E5430 processor (supporting both SSE2 and SSSE3) running at 2.66 GHz. The throughput results of various RAIDq implementations are shown in Table 3. *Here, the decoding throughput is measured with maximal erasures in the data blocks and hence represents a worst-case scenario.* Table 4 shows the performance of detecting and correcting from two random errors for RAIDq 8 base on the $[1,\alpha,\alpha^{1/2},X]$ Plank's code. Such error correcting is *not yet* a standard feature and is usually done by a long-running process in the background ("data scrubbing"), so the performance requirement might be not so stringent. Lastly, we note that a slight mismatch in encoding throughput between Table 3 and 4 is possibly due to measurement error.

4.2 Comparison with Reed-Solomon RAID

From Table 3, we see that RAIDq almost always outperforms Reed-Solomon RAID. Here we analyze such performance gain at an algorithmic level. The main difference is that Reed-Solomon encoding involves general finite-field multiplication, whereas the encoding of Plank's codes only involves multiplication by several fixed elements in the field, which can be optimized by the techniques described in Section 3. RAIDq also has an edge over Reed-Solomon RAID even without the above optimizations because RAIDq always uses an `XOR` checksum, which is not possible with general Reed-Solomon codes. Furthermore, the performance gain from encoding will affect decoding and error correcting because optimized decoders often reuse encoders as well, as described in Section 2.3. There are rare cases where RAIDq is outperformed by Reed-Solomon RAID, e.g., in decoding of $[1,\alpha,\alpha^{1/3},\alpha^{2/3}]$ and $[1,\alpha,\alpha^{1/2},X]$. This happens because here the decoder needs to perform additional expensive multiplications in $\mathbb{F}_{2^{16}}$ and $\mathbb{F}_{2^{24}}$ in order to support more data disks.

Finally, we note that all performance results presented here represent worst-case scenarios. *We expect that in practice, the most common disks failures can be recovered from the much faster RAID 5 or 6 checksum computation, which is possible because RAIDq includes them as special cases.*

References

[1] ANVIN, H. P. The mathematics of RAID-6, 2009.

[2] BERNSTEIN, D. J. Fast multiplication and its applications. *Algorithmic number theory 44* (2008), 325–384.

[3] CHEN, A. I.-T., CHEN, M.-S., CHEN, T.-R., CHENG, C.-M., DING, J., KUO, E. L.-H., LEE, F. Y.-S., AND YANG, B.-Y. SSE implementation of multivariate PKCs on modern x86 CPUs. In *CHES 2009* (Lausanne, Switzerland, September 2009), pp. 33–48.

[4] LEVENTHAL, A. Triple-parity RAID and beyond. *Queue 7*, 11 (December 2009), 30:30–30:39.

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

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

[7] PLANK, J. S., GREENAN, K. M., AND MILLER, E. L. Screaming fast Galois Field arithmetic using Intel SIMD instructions. In *FAST 2013* (San Jose, CA, USA, February 2013).

[8] PLANK, J. S., LUO, J., SCHUMAN, C. D., XU, L., AND WILCOX-O'HEARN, Z. A performance evaluation and examination of open-source erasure coding libraries for storage. In *FAST 2009* (San Francisco, CA, USA, February 2009).