Screaming Fast Galois Field Arithmetic Using Intel SIMD Instructions

James S. Plank
EECS Department
University of Tennessee

Kevin M. Greenan
EMC Backup Recovery
Systems Division

Ethan L. Miller
Computer Science Department
UC Santa Cruz

Abstract

Galois Field arithmetic forms the basis of Reed-Solomon and other erasure coding techniques to protect storage systems from failures. Most implementations of Galois Field arithmetic rely on multiplication tables or discrete logarithms to perform this operation. However, the advent of 128-bit instructions, such as Intel's Streaming SIMD Extensions, allows us to perform Galois Field arithmetic much faster. This short paper details how to leverage these instructions for various field sizes, and demonstrates the significant performance improvements on commodity microprocessors. The techniques that we describe are available as open source software.

1 Introduction

Storage systems rely on erasure codes to protect themselves from data loss. Erasure codes provide the basic underlying technology for RAID-6 systems that can tolerate the failures of any two disks [1, 7], cloud systems that tolerate larger numbers of failures [6, 12, 19, 21], and archival systems that tolerate catastrophic situations [18, 29, 31, 32]. The canonical erasure code is the Reed-Solomon code [27], which organizes a storage system as a set of linear equations. The arithmetic of these linear equations is *Galois Field arithmetic*, termed $GF(2^w)$, defined as arithmetic over w-bit words. When applied to a storage system, encoding and decoding compute these linear equations by multiplying large regions of bytes by various w-bit contants in $GF(2^w)$ and then combining the products using bitwise exclusive-or (XOR). Thus, the two fundamental operations are performing multiplication of a region of bytes by a constant, and performing the XOR of two regions of bytes.

Historically, implementations of Galois Field arithmetic use multiplication tables for small values of w, logarithm tables for larger values, and incremental shifters for even larger values [9, 21, 22]. In practice, the performance of multiplication is at least four times slower than XOR [26].

In recent years, processors that implement Intel's Streaming SIMD Extensions instruction set have become ubiquitous. The SIMD instructions allow 128-bit numbers to be manipulated in the CPU, and their ramifications for multiplying regions of numbers by constants in $GF(2^w)$ are significant. Anecdotes of performing Galois Field arithmetic "at cache line speeds" have become commonplace at meetings such as Usenix FAST. However, as these anecdotes have typically come from those working at storage companies, the exact mechanics of using the SIMD instructions have been guarded.

This short paper details the SIMD instructions to multiply regions of bytes by constants in $GF(2^w)$ for $w \in \{4, 8, 16, 32\}$. These are the most common values of w in storage installations (please see Section 2 for a discussion of why w matters in a storage system). Each value of w requires different implementation techniques, some of which are subtle. We present each technique in enough detail for a reader to implement it in his or her own storage system, and we detail an open source implementation. The performance of these implementations is 2.7 to 12 times faster than other implementations of Galois Field arithmetic.

2 Erasure Codes and Galois Fields

Erasure codes that are based on Galois Field arithmetic are defined by a set of linear equations. A storage system composed of n disks is partitioned into k that hold data and m that hold coding information that is calculated from the data. For the purposes of the code, each disk logically stores one w-bit word. Suppose the words on the data disks are labeled $d_0, \ldots, d_{k-1}$ and the words on the coding disks are labeled $c_0, \ldots, c_{m-1}$. Then creating the coding words from the data words may be ex-

pressed by m equations (where arithmetic is in $GF(2^w)$:

$$\text{For } 0 \leq i < m: \quad c_i = \sum_{j=0}^{k-1} a_{i,j} d_j. \tag{1}$$

For example, the RAID-6 installation in the Linux kernel has two coding disks, c_0 and c_1, which are created by two equations [1]:

$$\begin{aligned} c_0 &= d_0 + d_1 + d_2 + \ldots + d_{k-1} \\ c_1 &= d_0 + 2d_1 + 4d_2 \ldots + 2^{k-1} d_{k-1} \end{aligned}$$

When up to m disks fail, we are left with m equations with up to m unknown variables. We decode by solving those equations with Gaussian elimination or matrix inversion.

There are a variety of erasure codes that are organized in this fashion. The most prevalent one is the Reed-Solomon code [27], which employs a generator matrix to define the above equations for any value of k and m, so long as $k + m \leq 2^w$. Reed-Solomon codes are *Maximum Distance Separable (MDS)*, which means that they tolerate the loss of any m disks. There is a detailed tutorial by Plank that spells out exactly how to implement Reed-Solomon codes in storage systems [22, 24].

More recently, other erasure codes have been developed that are based on the above methodology. Pyramid codes [11], LRC codes [12] and F-MSR codes [10] are all based on Galois Field arithmetic to achieve improved encoding performance, tolerance to sector failures, and improved decoding performance. LRC codes are the erasure code currently employed by the Microsoft Azure cloud storage system [12].

When implementing an erasure code, a value of w must be selected. This has three main impacts. First, it impacts the size of the storage system. For example, when using Reed-Solomon codes, a value of $w = 4$ can be used on systems up to 16 disks in size. A value of $w = 8$ expands that to 256 disks, and a value of $w = 16$ expands that to 65,536 disks. Second, it impacts layout. Implementors typically choose values of w that are factors of two so that w-bit words in the coding system fit precisely into machine words. Finally, it impacts performance. We will explore performance below, but the rule of thumb is that larger values of w perform more slowly than smaller values. For that reason, one typically chooses the smallest value of w such that w is a power of two and has the properties needed for the size of one's storage system. For example, Azure uses $w = 4$ [12], while Cleversafe uses $w = 8$ [28]. There are benefits to larger w that justify the extra complexity of implementation. For example, HAIL blends security and erasure coding, using large values of w such as 32 and 64 [21].

3 XOR's and Region Multiplication

Although coding equations such as (1) work on single words, in reality, the data and coding words are larger regions of bytes, such as disk sectors in RAID systems, or very large blocks of sectors in cloud systems [12, 17]. The reason is that when one partitions a region of bytes into multiple words, one can perform operations on them in parallel.

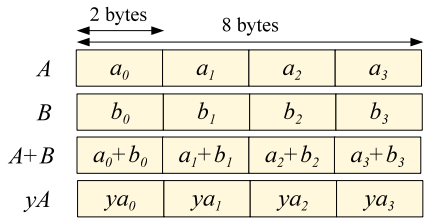

Figure 1: Two 8-byte regions of memory A and B are partitioned into four 16-bit words. The last two lines show how addition and multiplication are mapped to the individual words of the regions.

We depict an example in Figure 1. Here we have two 8-byte regions of memory, A and B, each of which is partitioned into four 16-bit words, $a_0, \ldots, a_3$ and $b_0, \ldots, b_3$. With Galois Field arithmetic, addition is equivalent to XOR; thus adding the region A and B is equivalent to adding each a_i with each b_i, and it may be implemented with a single 64-bit XOR operation. One may view multiplication of A by a constant y as multiplying each a_i by y. Performing this operation fast is the subject of this paper.

4 Streaming SIMD Instructions

Intel's Streaming SIMD Instructions [15] have has become ubiquitous in today's commodity microprocessors. They are supported in CPUs sold by Intel, AMD, Transmeta and VIA. Compiler support for these instructions have been developed as well; however, leveraging these instructions for Galois Field arithmetic requires too much application-specific knowledge for the compilers.

The basic data type of the SIMD instructions is a 128-bit word, and we leverage the following instructions in our implementations:

- *mm_set1_epi8(b)* creates a 128-bit variable by replicating the byte b sixteen times. *mm_set1_epi16(b)*, *mm_set1_epi32(b)* and *mm_set1_epi64(b)* set the variable by replicating 2-byte, 4-byte and 8-byte words respectively.
- *mm_and_si128(a, b)* and *mm_xor_si128(a, b)* perform bitwise AND and bitwise XOR on 128-bit words a and b respectively.

byte	f	e	d	c	b	a	9	8	7	6	5	4	3	2	1	0
table1:	0b	0c	05	02	04	03	0a	0d	06	01	08	0f	09	0e	07	00
table2 = mm_slli_epi64(table1, 4):	b0	c0	50	20	40	30	a0	d0	60	10	80	f0	90	e0	70	00
mask1 = mm_set1_epi8(0xf):	0f	0f	0f	0f	0f	0f	0f	0f	0f	0f	0f	0f	0f	0f	0f	0f
mask2 = mm_set1_epi8(0xf0):	f0	f0	f0	f0	f0	f0	f0	f0	f0	f0	f0	f0	f0	f0	f0	f0
A:	39	1d	9f	5a	aa	ab	15	c3	63	e0	7c	43	fb	83	16	23
l = mm_and_si128(A, mask1):	09	0d	0f	0a	0a	0b	05	03	03	00	0c	03	0b	03	06	03
l = mm_shuffle_epi8(l, table1):	0a	05	0b	03	03	04	08	09	09	00	02	09	04	09	01	09
h = mm_and_si128(A, mask2):	30	10	90	50	a0	a0	10	c0	60	e0	70	40	f0	80	10	20
h =mm_srli_epi64(h, 4):	03	01	09	05	0a	0a	01	0c	06	0e	07	04	0f	08	01	02
h = mm_shuffle_epi8(h, table2):	90	70	a0	80	30	30	70	20	10	c0	60	f0	b0	d0	70	e0
yA = mm_xor_si128(h, l):	9a	75	ab	83	33	34	78	29	19	c0	62	f9	b4	d9	71	e9

Figure 2: Multiplying a 128-bit region A by $y = 7$ in $GF(2^4)$. The first few instructions show variables that are set up before performing the multiplication. The last six perform the 32 multiplications using two table lookups.

- *mm_srli_epi64(a, b)* treats a as two 64-bit words, and right shifts each by b bits. *mm_slli_epi64(a, b)* performs left shifts instead.
- *mm_shuffle_epi8(a, b)* is the real enabling SIMD instruction for Galois Fields. Both a and b are 128-bit variables partitioned into sixteen individual bytes. The operation treats a as a 16-element table of bytes, and b as 16 indices, and it returns a 128-bit vector composed of 16 simultaneous table lookups, one for each index in b.

5 Calculating yA in $GF(2^4)$

When $w = 4$, there are only 16 values that a word may have. Thus, employing a 16×16 multiplication table requires very little memory, and such a table may be precomputed very quickly. While that suffices to perform multiplication of single values, it is less efficient when performing yA where y is a 4-bit word and A is a region of bytes, because a separate table lookup is required for every four bits in A.

The *mm_shuffle_epi8(a, b)* instruction may be leveraged so that yA may be performed 128-bits at a time with just two table lookup operations. We give a detailed example in Figure 2. In this example, we are multiplying a 16-byte region, A, by $y = 7$ in $GF(2^4)$. In the figure, we show 128-bit variables on the right and the instructions that create them on the left. The variables are presented as 16 two-digit numbers in hexadecimal. The figure is broken into three parts. The first part displays two multiplication tables that are created from y, and two masks. The second part shows the 16 bytes of A. The third part shows how to implement the multiplication. The low-order four bit words of each byte of A are isolated using the first mask, and they are used to perform 16 simultaneous table lookups that multiply those words by 7. The result is put into l. The high-order four bit words of each byte of A are then isolated using the second mask, right shifted by four bits and then used to perform table lookup in the second table. The result is put into h, which is combined with l to create the product. Thus, six instructions suffice for the 32 multiplications.

6 Calculating yA in $GF(2^8)$

As with $GF(2^4)$, one can implement multiplication in $GF(2^8)$ with table lookup. For a given value of y, one needs a 256-element table of bytes to look up each potential value of ya_i. However, *mm_shuffle_epi8()* only works on 16-element tables. To keep the prose clean, let us first drop the "i" subscript of a_i. To leverage *mm_shuffle_epi8()*, we observe that we can split a into two four-bit words, a_l and a_r, and then perform the multiplication with two table lookups. This is called a "left-right table" by Greenan *et al* [9].

To be precise, let a be an 8-bit word and let a_l and a_r be four-bit words so that:

$$a = (a_l \ll 4) \oplus a_r.$$

Then:

$$ya = y(a_l \ll 4) \oplus ya_r.$$

Thus, to implement multiplication, we create *table1* and *table2* as in Section 5. *Table1* contains the product of y with all four-bit words, a_r, and *table2* contains the product of y with all eight-bit words whose last four bits are zeros, $(a_l \ll 4)$, indexed by a_l. We give an example of these tables when $y = 7$ in Figure 3. Unlike Figure 2,

these tables are not four-bit shifts of each other. Instead, each table contains the eight bit product of 7 with another element of $GF(2^8)$ whose first or last four bits happen to equal zero. For example, in $GF(2^8)$, the product of 7 and 0xa is 0x36, and the product of 7 and 0xa0 is 0x47.

byte	f	e	d	c	b	a	9	8	7	6	5	4	3	2	1	0
table1:	2d	2a	23	24	31	36	3f	38	15	12	1b	1c	09	0e	07	00
table2:	ea	9a	0a	7a	37	47	d7	a7	4d	3d	ad	dd	90	e0	70	00

Figure 3: The two tables for multiplying 128-bit regions by $y = 7$ in $GF(2^8)$.

After the two tables are constructed, the product yA is calculated with the exact same six instructions as in Figure 2. Since the tables are small, and there are only 256 potential values of y, they may be precomputed.

It should be noted that this implementation technique was documented in assembly code by Anvin in his explanation of RAID 6 decoding in the Linux kernel [1].

7 Calculating yA in $GF(2^{16})$

A similar technique may be employed for $GF(2^{16})$; however, there are quite a few subtleties. As above, so that we can use *mm_shuffle_epi8()*, we split each a into subwords whose sizes are four bits. Let those words be a_0 through a_3. Then we calculate the product with the equation:

$$ya_i = y(a_3 \ll 12) + y(a_2 \ll 8) + y(a_1 \ll 4) + ya_0.$$

Unfortunately, we cannot map each subproduct to a single *mm_shuffle_epi8()* instruction, because each subproduct is a 16-bit word, and *mm_shuffle_epi8()* only performs table lookup on bytes. Thus, we need to use two tables per subproduct: one that holds the low-order bytes of each product, and one that holds the high-order bytes. This is a total of eight tables, which we label Ti_{high}, for the tables that calculate the high product bytes using a_i, and Ti_{low}, for the tables that calculate the low bytes using a_i.

To fully utilize *mm_shuffle_epi8()'s* ability to perform 16 simultaneous table lookups, it is best *not* to map words to contiguous memory locations as in Figure 1. Instead, every set of 16 words is mapped to two 128-bit variables, where the high byte (a_3 and a_2) of each word is in the first variable, and the low byte (a_1 and a_0) is in the second variable.

Because this can be quite confusing, we show a picture in Figure 4, which shows the first six bytes of two 128-bit variables, A_{high} and A_{low}. The variables together hold sixteen 16-bit words, of which we show three, b, c and d, in the picture. The high bytes of b, c and d are held in A_{high} and the low bytes are held in A_{low}.

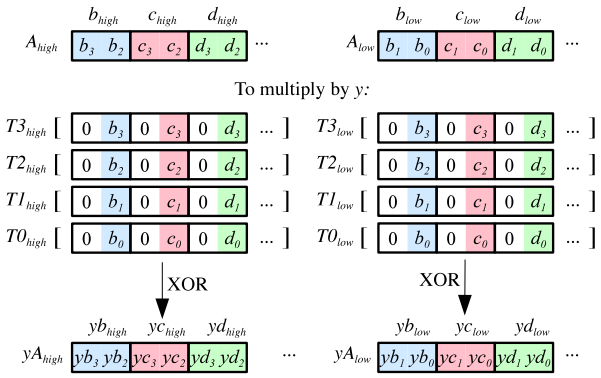

Figure 4: Using eight lookup tables and an alternate mapping of words to memory, to multiply a region A of sixteen 16-bit words in $GF(2^{16})$. Each rectangle is a byte, and only three words, b, c and d are shown. Each word is split into two bytes, one of which is stored in A_{high} and one in A_{low}. Each product yA_{high} and yA_{low} requires four *mm_shuffle_epi8()* operations, plus some bit masks and shifts.

Calculating the products requires four *mm_shuffle_epi8()* operations for each of yA_{high} and yA_{low}. For yA_{high}, we use the tables Ti_{high} and for yA_{low}, we use the tables Ti_{low}. By splitting each 16-bit word into two 128-bit variables, we can fully utilize *mm_shuffle_epi8()*'s ability to perform 16 simultaneous table lookups.

We call this technique *Altmap*, because it maps words to memory differently than the standard mapping of Figure 1. The down sides to *Altmap* are that memory regions are constrained to be multiples of 32 bytes, and that it is more confusing than the standard mapping. For the purposes of erasure coding, however, memory regions are only multiplied by constants and XOR'd together, so one only needs to actually extract the 16-bit values from the memory regions when debugging.

The eight tables consume a total of 128 bytes, and there are 64K potential values of y, so it is conceivable that the tables may be precomputed and stored. However, computing them at the beginning of the multiplication operation takes under 200 instructions, which means that table creation may be amortized by multiplying larger regions of memory by the same constant.

7.1 Using the Standard Mapping

If the standard mapping must be employed, for example to be interoperable with other erasure coding implementations that employ the standard mapping, then

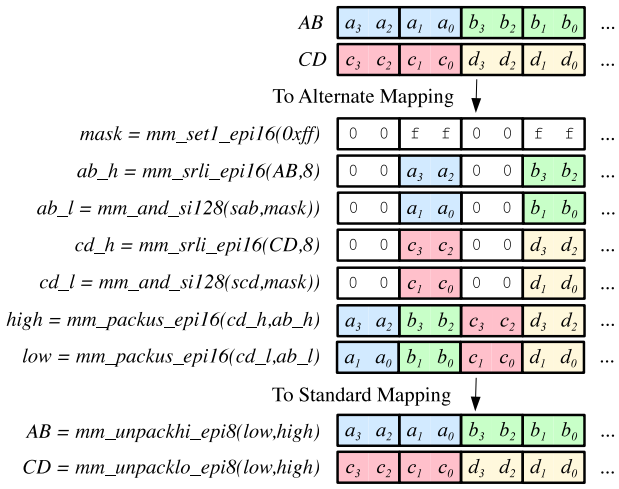

Figure 5: The SIMD instructions for converting two 128-bit variables to the alternate mapping and back again.

one may convert the standard mapping to the alternate mapping before multiplying each set of 16 words, and then convert back. The conversions leverage the instructions *mm_packus_epi16()*, *mm_unpackhi_epi8()* and *mm_unpacklo_epi8()*, which pack two 128-bit variables into one, and unpack one 128-bit variable into two. We demonstrate the operations in Figure 5, which pictures the first four bytes of each 128-bit variable.

8 Calculating yA in $GF(2^{32})$

The same technique for $w = 16$ generalizes to larger values of w. For $w = 32$, we partition each word a into eight four-bit words which are be used as indices for *mm_shuffle_epi8()* calls. Since the products are 32-bit words, there must be four tables for each four-bit word, for a total of 32 different lookup tables. As with $w = 16$, we employ the alternate mapping so that each set of sixteen 32-bit words is held in four 128-bit vectors in such a way that each vector holds one byte from each word. We may convert the standard mapping to the alternate mapping and back again using a similar technique as with $w = 16$.

9 Performance

We have released an open source library in C called GF-Complete [25]. The library implements all documented techniques for performing Galois Field arithmetic, including logarithm and multiplication tables from standard libraries like **jerasure** [26] and Rizzo's FEC library [30], split multiplication tables [26, 9], composite fields [9, 21], bit-grouping tables [21] and Anvin's technique based on fast multiplication by two, which is included in the Linux kernel and has received further attention that leverages GPU co-processors [1, 16].

Our performance evalution briefly demonstrates the dramatic speed improvements due to these techniques. GF-Complete includes a performance testing module which fills regions of bytes with random values, and then multiplies those regions by contants in $GF(2^w)$. We show results of the performance tester in Figure 6. The testing machine is a 3.4 GHz Intel Core i7-3770 with 16 GB of DRAM, a 256 KB L2 cache and an 8 MB L3 cache. In the tests, we vary the region sizes from 1 KB to 1 GB in multiples of four and encode a total of 5 GB.

On the rightmost graph, we include baseline performance numbers for reference: the speed of **memcpy()**, the speed of *mm_xor_si128()*, and the speed of multiplying regions by two using Anvin's SIMD optimization from the Linux kernel [1], which is independent of w.

For controls, we include historically "standard" techniques as implemented by **jerasure**/Rizzo [26, 30]: Multiplication tables for $w \in \{4, 8\}$, logarithm tables for $w = 16$ and seven "split" multiplication tables, each of size 256x256 for $w = 32$. For $w \in \{4, 8, 16\}$, we also include "16-bit tables." These break up the regions into 16-bit words and use them as indices into a table that is created at the beginning of multiplication (except for $w = 4$, where the tables are precomputed). This improves upon the standard multiplication tables for $w \in \{4, 8\}$ because 16 bits of table-lookup are performed at a time, as opposed to four table lookups for $w = 4$ and two for $w = 8$. We also include "By-two," which structures multiplication by selectively multiplying 128-bit regions by two using Anvin's optimization [1, 16].

The SIMD implementations and the baselines follow similar trajectories which are cache dependent. As the region sizes grow, performance improves as setup costs are amortized. Performance reaches a peak as caches are saturated: **Memcpy()** and XOR are limited by the L1 cache; while the SIMD techniques and Anvin's optimization are limited by the L2 cache. When the L2 cache is saturated (at a region size of 256 KB), performance drops slightly. When the L3 cache is saturated (at a region size of 4 MB), the performance drops dramatically. In fact, then the L3 cache is saturated, the SIMD performance, even at $w = 32$, matches that of XOR.

Because they perform roughly the same operations, $w = 4$ and $w = 8$ perform identically. $w = 16$ performs slightly slower, and $w = 32$ slower still. With respect to the controls, at their peak, the SIMD implementations perform 2.7 to to 12 times faster than the other implementations. When $w \in \{16, 32\}$, the alternate mappings perform 48 and 33 percent faster than the standard mappings, which require conversion to the alternate mapping and back again.

The base conclusion to draw from the tests in Figure 6

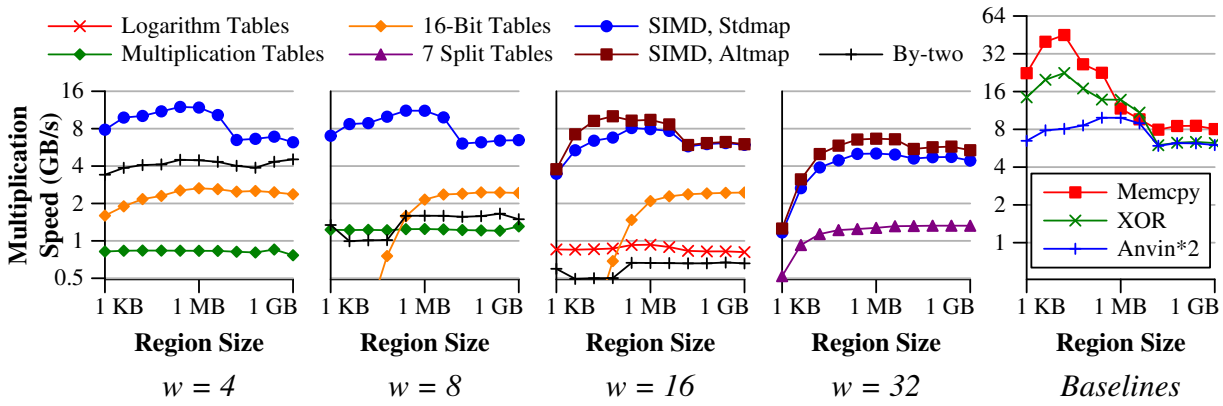

Figure 6: The performance of multiplying regions of bytes by constants in $GF(2^w)$.

is that with the SIMD instructions, the cache becomes the limiting factor of multiplication, and I/O becomes the dominant concern with erasure coding.

10 Related Work

There has been quite a lot of work on implementing Galois Field arithmetic in software. The various implementation techniques detailed in Section 9 are explained in papers by Anvin [1], Greenan [9], Kalcher [16], Lopez [20], Luo [21] and Plank [22]. The implications on erasure coding systems has been demonstrated to varying degrees by Greenan [9], Hu [10], Huang [12], Khan [17] and Luo [21]. The techniques in this paper have been leveraged to sustain throughputs of over 4 GB/s in recent tests of Reed-Solomon coding [23].

Of particular interest is the judicious design of RAID 6 in the Linux kernel [1]. By employing the encoding equations c_0 and c_1 from Section 2, one coding disk requires only XOR calculations, and the second requires XOR's and multiplications by two using the baseline implementation from Figure 6. General-purpose multiplication is only required upon decoding, and the kernel employs a technique which is identical to our technique for $GF(2^8)$, albeit implemented directly in assembly code. As such, the CPU performance of encoding and decoding is thoroughly optimized.

This paper focuses solely on multiplying regions of bytes by constants. Its results do not apply to optimizing single multiplication operations. For these, the standard tables work the best for $w \leq 16$, and the grouping techniques of Luo perform the best for $w = 32$ [21].

While we only employ Intel's SIMD instructions, other instructions sets like Altivec and AVX2 feature permutation instructions that may be leveraged similarly. The Intel Performance Primitives library [14] has deprecated (as of revision 7.1) Galois Field operations; however, these only support single multiplications for $GF(2^8)$ and cannot be leveraged in a SIMD fashion.

11 Conclusion

In this paper, we demonstrate how to leverage 128-bit SIMD instructions to perform multiplication of a region of bytes by a constant in Galois Fields $GF(2^w)$, where $w \in \{4, 8, 16, 32\}$. For $w \in \{16, 32\}$, an alternate mapping of words to memory allows us to further optimize performance. For a small penalty, one may convert this mapping back to the standard mapping.

We have implemented these techniques in an open-source library whose performance we have tested and compared to other Galois Field implementations. The improvement ranges from 2.7 to 12 times faster than the traditional implementations, and helps to perpetuate a trend of worrying more about I/O than CPU performance in erasure coding settings.

The speed of multiplication using these techniques is so much faster than previous implementations that it has implications on the design of erasure codes. A previous assumption in erasure code design has been that erasure codes based on XOR operations are significantly faster than those based on Galois Field arithmetic. This assumption has led to the design of many XOR-only erasure codes [2, 4, 5, 7, 13, 33]. When Galois Field multiplication is cache-limited, erasure codes based on Galois Field arithmetic become viable alternatives to XOR codes. Recent examples of codes based on Galois Field arithmetic are LRC and Rotated Reed-Solomon codes [12, 17] for improved recovery performance in cloud storage systems, regenerating codes to lower network bandwidth in decentralized storage settings [8, 10], and PMDS/SD codes to improve storage efficiency in disk arrays [3, 23]. The applicability of these codes is heightened by the techniques in this paper, and we anticipate that future code design will rely more on Galois Field arithmetic than on XOR's.

12 Acknowledgements

This material is based upon work supported by the National Science Foundation under grants CNS-0917396, IIP-0934401 and CSR-1016636. The authors would like to thank shepherd Joseph Tucek and the FAST referees for constructive comments on the paper.

References

[1] H. P. Anvin. The mathematics of RAID-6. `http://kernel.org/pub/linux/kernel/people/hpa/raid6.pdf`, 2011.

[2] 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.

[3] M. Blaum, J. L. Hafner, and S. Hetzler. Partail-MDS codes and their application to RAID type of architectures. IBM Research Report RJ10498 (ALM1202-001), February 2012.

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

[5] 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.

[6] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, J. Haridas, C. Uddaraju, H. Khatri, A. Edwards, V. Bedekar, S. Mainali, R. Abbasi, A. Agarwal, M. Fahim ul Haq, M. Ikram ul Haq, D. Bhardwaj, S. Dayanand, A. Adusumilli, M. McNett, S. Sankaran, K. Manivannan, and L. Rigas. Windows Azure Storage: A highly available cloud storage service with strong consistency. In *23rd ACM Symposium on Operating Systems Principles*, October 2011.

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

[8] 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.

[9] 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.

[10] Y. Hu, H. C. H. Chen, P. P. C. Lee, and Y. Tang. NCCloud: Applying network coding for the storage repair in a cloud-of-clouds. In *FAST-2012: 10th Usenix Conference on File and Storage Technologies*, San Jose, February 2012.

[11] C. Huang, M. Chen, and J. Li. 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] 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.

[13] 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.

[14] Intel Corporation. Intel Integrated Performance Primitives for Intel architecture, reference manual IPP 7.1. `http://software.intel.com`, 2000-2012.

[15] Intel Corporation. Intel 64 and IA-32 architectures software developers manual, combined volumes: 1, 2A, 2B, 2C, 3A, 3B and 3C. Order Number: 325462-044US, August 2012.

[16] S. Kalcher and V. Lindenstruth. Accelerating Galois Field arithmetic for Reed-Solomon erasure codes in storage applications. In *IEEE International Conference on Cluster Computing*, 2011.

[17] 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.

[18] A. Leventhal. Triple-parity RAID and beyond. *Communications of the ACM*, 53(1):58–63, January 2010.

[19] X. Li, A. Marchant, M. A. Shah, K. Smathers, J. Tucek, M. Uysal, and J. J. Wylie. Efficient eventual consistency in Pahoehoe, an erasure-coded key-blob archive. In *DSN-10: International Conference on Dependable Systems and Networks*, Chicago, 2010. IEEE.

[20] J. Lopez and R. Dahab. High-speed software multiplication in f_{2^m}. In *Annual International Conference on Cryptology in India*, 2000.

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

[22] 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.

[23] 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.

[24] J. S. Plank and Y. Ding. Note: Correction to the 1997 tutorial on reed-solomon coding. Technical Report CS-03-504, University of Tennessee, April 2003.

[25] 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.

[26] 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 *FAST-2009: 7th Usenix Conference on File and Storage Technologies*, pages 253–265, February 2009.

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

[28] J. K. Resch and J. S. Plank. AONT-RS: blending security and performance in dispersed storage systems. In *FAST-2011: 9th Usenix Conference on File and Storage Technologies*, pages 191–202, San Jose, February 2011.

[29] S. Rhea, P. Eaton, D. Geels, H. Weatherspoon, B. Zhao, and J. Kubiatowitz. Pond: The OceanStore prototype. In *FAST-2003: 2nd Usenix Conference on File and Storage Technologies*, San Francisco, January 2003.

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

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

[32] M. W. Storer, K. M. Greenan, E. L. Miller, and K. Voruganti. POTSHARDS – a secure, long-term storage system. *ACM Transactions on Storage*, 5(2), June 2009.

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