A Complete Treatment of Software Implementations of Finite Field Arithmetic for Erasure Coding Applications

James S. Plank Kevin M. Greenan Ethan L. Miller

University of Tennessee Technical Report UT-CS-13-717
October 14, 2013
http://web.eecs.utk.edu/~plank/plank/papers/UT-CS-13-717.html

This paper has been submitted for publication. Please see the web site above for the current status of the paper.

GF-Complete is maintained at https://bitbucket.org/jimplank/gf-complete.
Jerasure, which is built on GF-Complete, is maintained at https://bitbucket.org/jimplank/jerasure.

Abstract

Finite field arithmetic lies at the heart of erasure codes that protect storage systems from failures. This arithmetic defines addition and multiplication over a closed set of numbers such that every number has a unique multiplicative inverse. For storage systems, the size of these sets is typically a power of two, and the finite fields most often employed are Galois Fields, denoted $GF(2^w)$. There are many ways to implement finite field arithmetic in software, plus a few tricks to aid performance. In this paper, we describe the various implementations and tricks, in tutorial-level detail, as they specifically apply to erasure coded storage systems. We also introduce an open-source Galois Field field arithmetic library called "GF-Complete" that implements all of these techniques and tricks, and give a rough performance evaluation.

This work is supported by the National Science Foundation, under grants CNS-0917396, IIP-0934401, CSR-1016636, REU Supplement CSR-1128847, and research awards by Google and IBM.

Author's addresses: J. S. Plank, EECS Department, University of Tennessee, Knoxville, TN 37991; K. M. Greenen, Box, Inc., 4440 El Camino Real, Los Altos, CA 94022; E. L. Miller, Computer Science Department, Baskin School of Engineering, University of California, 1156 High Street, MS SOE3, Santa Cruz, CA 95064.

1 Introduction

Erasure codes provide fault-tolerance for nearly all of today's storage systems, from RAID [1, 12, 11] to cloud [3, 9, 28] and archival storage systems [29, 32]. Most erasure codes, such as the well-known Reed-Solomon erasure codes, are based on finite field arithmetic [27]. A typical erasure coding scenario is depicted in Figure 1. In this scenario, k pieces of data, which are typically large buffers of bytes, are stored in such a way that they fail independently. This can be on a secondary storage medium like disk, or in physically disparate primary storage media. These pieces of data are encoded onto m additional storage buffers, which are called "coding" buffers. If the erasure code is "Maximum Distance Separable (MDS)," data may be recalculated from any k of the $n = k+m$ storage buffers; thus, loss of any m buffers causes no data loss as long as the position of the missing buffers is known (*i.e.*, i is known for all missing D_i).

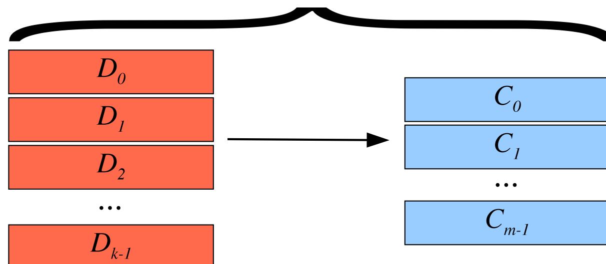


Figure 1: Typical erasure coding scenario, where k large storage buffers are encoded onto m additional storage buffers such that the failure of up to m of the $n = k + m$ buffers may be tolerated.

Many erasure codes, including the very popular Reed-Solomon codes, are defined to work on finite sets of numbers. These are often called *words* or *symbols*, and are stored in binary as w-bit numbers. The erasure code logically encodes k words onto m words in such a way that the loss of any m words (of the $k+m$ total words) may be tolerated without data loss. When such a code is implemented in a real storage system, each storage buffer, D_i and C_j, is typically partitioned into words, $d_{i,0}, d_{i,1}, \ldots$ and $c_{j,0}, c_{j,1}, \ldots$, and each collection of one word from each storage region ($d_{i,x}$ and $c_{j,x}$) is encoded together. Because of this partitioning, the number of bits per word, w, is constrained so that words fit easily into machine words, and collections of words fit neatly into blocks. As a result, w is typically a power of two; in other words, $w \in \{4, 8, 16, 32, 64, 128\}$ for most codes.

The choice of w is typically dictated by parameters of the coding system. For example, Reed-Solomon codes only have the MDS property when $n \leq 2^w + 1$ [18]. Regenerating codes require w to be big enough that the probability of non-regeneration is extremely small [5]. The HAIL storage system employs $w \geq 32$ to blend security and fault-tolerance [15]. Code designers typically attempt to minimize w, because implementations of small values of w perform faster than those of large values [24].

A standard depiction of erasure codes is drawn in Figure 2. The erasure code is expressed as a matrix-vector product, where a Generator matrix is applied to a vector composed of the k words of data to yield a "codeword" composed of the k data words and the m coding words. All elements of the system are w-bit words, and an MDS code guarantees that any m elements of the codeword may be lost without losing data.

Since all of the elements of the system are w-bit words, the matrix-vector product must be calculated using *finite field arithmetic*. This arithmetic defines addition and multiplication over the closed set of w-bit numbers such that each number has a unique multiplicative inverse. In other words, for each non-zero number i, there is a unique number j such that $i * j = 1$. The canonical arithmetic for these systems is Galois Field arithmetic, denoted $GF(2^w)$.

While the mechanics and properties of Galois Field arithmetic are well-known and well documented, there are a variety of ways in which these codes may be implemented in software, and there are a variety of implementation decisions that affect performance. The purpose of this paper is to present all of these implementation techniques and decision points, especially as they pertain to erasure coded storage systems. Additionally, we have implemented an open-source library in C for finite field arithmetic, called *GF-Complete*. This library includes all implementation techniques discussed in the paper and is available on the Internet for no charge. We evaluate the performance of this library on commodity processors so that potential users can have a rough idea of how the various implementations can work in storage installations.

Figure 2: Typical depiction of an erasure code as a matrix vector product, where all elements of the system are w-bit words.

2 Reference Material and Related Work

Much of the material in this paper is covered in greater depth by other sources. Standard texts on coding theory provide complete treatments of Galois Field arithmetic and Reed-Solomon coding for error and erasure correction [21, 18, 33, 19]. Plank's tutorial on Reed-Solomon coding [22] presents the material specifically for the storage practitioner, and provides details on using discrete logarithms to implement Galois Fields. Blömer, *et al.* use bit matrices to implement Galois Fields [2], and Greenan, *et al.* provide thorough discussions of table-based implementations and composite operations [6]. Techniques that apply effectively to large values of w are given detailed treatments by Lopez, *et al.* and Luo, *et al.*, and Anvin poineered the use of large computer words to perform multiple finite field operations of smaller words [14, 1, 15]. The use of SIMD instructions for fast region operations was first presented by Li and Huan-Yan [13], and then by Anvin [1]. It was given a more complete treatment by Plank, *et al.* [23]. Some techniques, such as the generalizations of the *SHIFT* technique, lazy generation of tables and the alternate mappings of field elements to memory are implementation techniques that are new to this work. The open source library, further consideration of vector instructions and performance evaluation are new as well.

3 The Requirements of Finite Field Arithmetic in Storage Systems

In order to realize the matrix-vector product of Figure 2, a finite field arithmetic implementation must perform addition and multiplication of w-bit numbers. When failures occur, the generator matrix is massaged into a decoding matrix, which typically requires a matrix inversion. To perform the inversion, one must be able to perform division of w-bit numbers, either directly or by multiplying by the multiplicative inverse of a number.

Additionally, real storage installations look more like Figure 3 than Figure 2. The identity matrix is managed implicitly, and the vector and product parts of the equation are not single numbers, but large storage buffers. Although they are partitioned into single numbers, it is typically much more efficient to add the buffers together as a whole, and to multiply whole buffers by single constants.

Thus, a finite field arithmetic library should support the following operations:

- **uint$_w$_t add(uint$_w$_t a, uint$_w$_t b);**

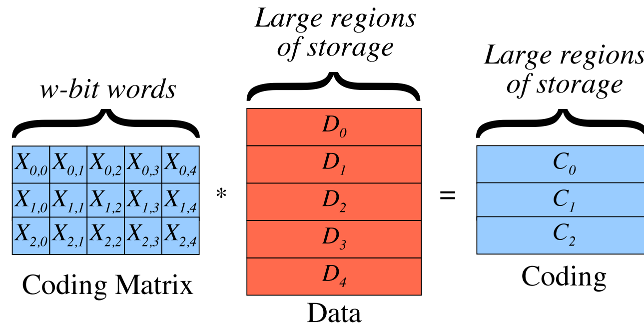


Figure 3: Erasure codes as they typically apply to real storage installations.

- **uint$_w$_t multiply(uint$_w$_t** a**, uint$_w$_t** b);
- **uint$_w$_t divide(uint$_w$_t** a**, uint$_w$_t** b);
- **void add_region(void** *a**, void** *b**, void** *sum**, int** $bytes$);
- **void multiply_region(uint$_w$_t** a**, void** *b**, void** *$product$**, int** $bytes$**, bool** add);

The parameter types **uint$_w$_t** are unsigned integer types represented as w-bit words, such as `uint8_t` or `uint64_t` in C, which represent unsigned 8 and 64-bit integers respectively. The *add* parameter of **multiply_region()** specifies whether the product region should simply be set to the product of a and b, or whether that product should be added to existing contents of $product$ using exclusive-or.

While the performance of all operations should be good, the truly important ones are the region operations. This is because the single operations are only employed for creating and massaging the generator matrix, both infrequent operations. The majority of time in erasure coding is spent in the region operations, as megabytes to gigabytes of data are typically encoded or decoded using a single matrix.

4 Galois Field Arithmetic Summarized: Bits and Polynomials

It would be convenient if modular arithmetic would suffice to implement finite fields, which is the case when the number of elements in the field is a prime number. For erasure codes, however, the number of elements is a power of two, which means that modular arithmetic cannot be employed because no even number has a multiplicative inverse.

Instead, we rely on Galois Fields, which we summarize here. A Galois Field $GF(2^w)$ has 2^w elements. Depending on the scenario, we represent each element in one of four ways:

- As a decimal number between 0 and $2^w - 1$.
- As a hexadecimal number between 0 and $2^w - 1$.
- As a binary number with w digits.
- As a polynomial of degree $w - 1$, whose coefficients are binary.

To determine an element's polynomial representation, one starts with the binary representation: the coefficient of the term x^i is equal to the i-th digit of the binary representation. This is called the *standard basis* representation [15]. Figure 4 shows sample representations in each format for the numbers 0, 1, 2, 5, 7, 10, 11 and 13 in $GF(2^4)$. We use these numbers as examples in the descriptions that follow.

Decimal	Hexadecimal	Binary	Polynomial
0	0x0	0000	0
1	0x1	0001	1
2	0x2	0010	x
5	0x5	0101	x^2+1
7	0x7	0111	x^2+x+1
10	0xa	1010	x^3+x
11	0xb	1011	x^3+x+1
13	0xd	1101	x^3+x^2+1

Figure 4: Examples of the four representations of some elements of $GF(2^4)$.

For those unfamiliar with polynomial representations, we stress that the polynomial is never evaluated. Rather, it is simply used as a representation because of its properties with respect to addition, multiplication and division. For an element a, we denote its polynomial representation as $a(x)$:

$$a = a(x) = \sum_{i=0}^{w-1} a_i x^i, \tag{1}$$

where a_i is equal to the i-th digit in the binary representation of a.

Addition of two elements in $GF(2^w)$ corresponds to addition of their polynomials, where coefficients are added modulo two. For example, 10 + 13 is equal to 7:

$$10 + 13 \quad = \quad (x^3 + x) + (x^3 + x^2 + 1) \quad = \quad x^2 + x + 1 \quad = \quad 7.$$

Thus, addition is conveniently equivalent to the bitwise XOR operation on the binary representation. It also has the property that addition equals subtraction, and any element added to itself equals zero.

Multiplication on the other hand is more complex. When we multiply two numbers a and b, we start by multiplying their polynomials $a(x)$ and $b(x)$. To differentiate this polynomial multiplication from other kinds of multiplication in this paper, we call it *carry-free multiplication* and denote it with the operator $\otimes$. If the product of a carry-free multiplication has a degree less than w, then multiplication in $GF(2^w)$ is equal to carry-free multiplication. For example, it is easy to see that the product of 2 and 5 in $GF(2^4)$ is equal to 10:

$$2 \otimes 5 = (x)(x^2 + 1) = x^3 + x = 10.$$

However, the carry-free multiplication can result in a polynomial with a larger degree. For example, consider multiplying 10 and 13:

$$
\begin{aligned}
10 \otimes 13 &= (x^3+x)(x^3+x^2+1) \\
&= x^3(x^3+x^2+1) + x(x^3+x^2+1) \\
&= (x^6+x^5+x^3) + (x^4+x^3+x) \\
&= x^6+x^5+x^4+x.
\end{aligned}
$$

When this happens, we *reduce* the product $p(x)$ to an equivalent polynomial $p'(x)$ whose degree is less than w using a special polynomial of degree w called the *irreducible polynomial*, IP. We will not dwell on the construction of IP. Its main property is that it cannot be factored into smaller polynomials, and that property of irreducibility is central to multiplication.

To reduce $p(x)$ to $p'(x)$, we take it *modulo* the irreducible polynomial. Formally, we find some polynomial $e(x)$ such that:

$$p(x) = e(x)IP(x) + p'(x).$$

The irreducibility of IP guarantees that $e(x)$ and $p'(x)$ always exist and are unique.

Algorithmically, we can find $p'(x)$ using the following simple steps:

Input: Polynomial $p(x)$
Output: Reduced polynomial $p'(x)$
while TRUE **do**
  $d = \text{degree}(p(x))$;
  if $d < w$ **then**
    $p'(x) \leftarrow p(x)$;
    return $p'(x)$;
  end
  $p(x) \leftarrow p(x) + x^{(d-w)}IP(x)$;   // Reduce the degree of $p(x)$, guaranteeing termination
end

For example, a irreducible polynomial for $GF(2^4)$ is x^4+x+1 [33]. We use this polynomial to reduce the product of 10 and 13 in the table below:

$p(x)$	d	$x^{d-w}IP(x)$	New $p(x) = p(x) + x^{d-w}IP(x)$
$x^6+x^5+x^4+x$	6	$x^6+x^3+x^2$	$x^5+x^4+x^3+x^2+x$
$x^5+x^4+x^3+x^2+x$	5	x^5+x^2+x	x^4+x^3
x^4+x^3	4	x^4+x+1	x^3+x+1

Thus, $p'(x) = x^3+x+1$, and the product of 10 and 13 is 11.

There are tables of irreducible polynomials for all practical values of w [33]. For the values of w that are important for erasure coding, we list irreducible polynomials in Figure 5. The polynomials denoted "standard" are ones that are employed in current popular libraries for Galois Field arithmetic [31, 20, 26]. The "alternate" polynomials are given only in hexadecimal, and have better properties than the standard ones in certain situations, as described in Section 6.2.

w	Standard irreducible polynomial for $GF(2^w)$	In hexadecimal	Alternate
4	$x^4 + x + 1$	0x13	
8	$x^8 + x^4 + x^3 + x^2 + 1$	0x11d	
16	$x^{16} + x^{12} + x^3 + x + 1$	0x1100b	0x1002d
32	$x^{32} + x^{22} + x^2 + x + 1$	0x400007	0xc5
64	$x^{64} + x^4 + x^3 + x + 1$	0x1b	
128	$x^{128} + x^7 + x^2 + x + 1$	0x 87	

Figure 5: Irreducible polynomials for $GF(2^w)$, for values of w that are practical for erasure coding. For $w \geq 32$, we omit the leading bit in the hexadecimal representation, because in practical implementations, the bit does not fit into the machine word used to implement the number.

5 Representing Elements and Basic Operations

The unsigned integer representation of a number may be employed conveniently for all four representations of elements in $GF(2^w)$. Basic computer bit operations can then implement some of the basic arithmetic operations in the field. For example, both **add()** and **add_region()** may be implemented by the exclusive-or operation (XOR), which is supported by all computer architectures, usually in units of 64 and up to 256 bits per instruction.

Multiplication by x^i is equivalent to bit-shifting a number left by i bits, and determining the value of a_i is equivalent to checking the value of the ith bit of the binary representation of $p(x)$. Thus, when we use Equation 2 to define carry-free multiplication, we may use standard computer bit operations to both multiply by x^i and reduce the product.

$$a \otimes b = \sum_{i=0}^{w-1} a_i(x^i b). \tag{2}$$

We demonstrate the multiplication of 10 and 13 in $GF(2^4)$ using the binary representations of the numbers in Figure 6. In this figure, and others that involve binary, we use C notation for left-shift $(<<)$ and right-shift $(<<)$, and $\oplus$ for exclusive-or.

Action	Bit operations on binary words	Result
Carry-free multiplication of 10 and 13	$(1101 << 3) \oplus (1101 << 1)$	1110010
First reduction step: adding $x^2IP(x)$	$1110010 \oplus (10011 << 2)$	111110
Second reduction step: adding $xIP(x)$	$111110 \oplus (10011 << 1)$	11000
Third reduction step: adding $IP(x)$	$11000 \oplus 10011$	1011

Figure 6: Multiplying 10 and 13 in $GF(2^4)$ using computer bit operations.

6 Implementations of multiply()

6.1 The *SHIFT* Implementation: Directly from the Definition

Armed with this information, we may implement the multiplication algorithm detailed in Section 4 above very simply. We call this implementation *SHIFT*; its implementation in C for $GF(2^8)$ is shown in Figure 7. We also include a detailed example of how the C code multiplies 230 and 178 to yield the product 248 in $GF(2^8)$.

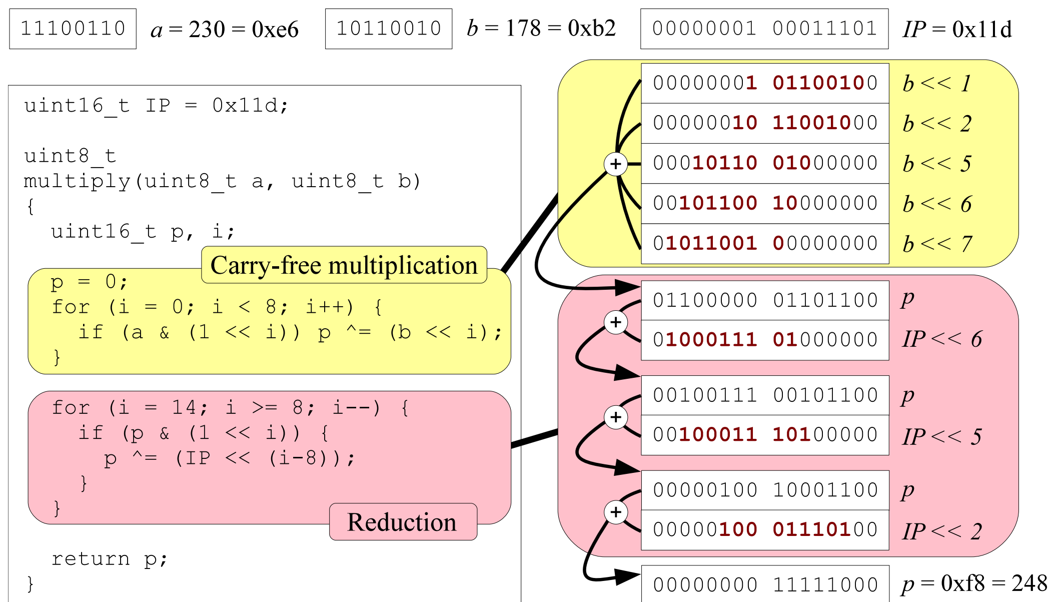


Figure 7: The *SHIFT* implementation in C of **multiply()** in $GF(2^8)$, plus an example of multiplying $a = 230$ and $b = 178$ to yield $p = 248$. The second **for** loop starts at 14 because the highest possible degree for $p(x)$ is 14.

Compared to the other implementations detailed below, this implementation has several practical problems. It requires $O(w)$ operations, and inconveniently, the product of the carry-free multiplication must be stored in a word whose size is $2w$ bits. Therefore, we do not recommend that this implementation be used in any setting where performance is a concern.

However, it is a good first step toward understanding many of the other implementations. Formally, when we multiply a and b, the *SHIFT* algorithm creates $p(x)$ with the following formula:

$$p(x) \quad = \quad \mathbf{reduce}(a \otimes b)$$

6.2 The *CARRY-FREE* Implementation: Using a carry-free multiplication instruction

Some computer architectures include an instruction that performs a carry-free multiplication. For example, Intel's `pclmulqdq` instruction performs a carry-free multiplication of two 64-bit numbers and places the result into a 128-bit vector. With this instruction, we can implement **multiply()** much more efficiently than with *SHIFT*. Its performance, however, depends on the structure of the irreducible polynomial.

We describe how it works, first with polynomials, and second with C code. Let us represent the irreducible polynomial as $x^w + pp(x)$, and let pp_d be equal to w minus the degree of $pp(x)$. For example, in $GF(2^4)$, $pp(x) = x + 1$ and $pp_d = 3$.

Let $p(x)$ be a polynomial whose degree is equal to $d \geq w$. We may represent p as the sum of two polynomials:

$$p(x) = p_{high}(x)x^w + p_{low}(x),$$

where $p_{high}(x)$ and $p_{low}(x)$ have degrees less than w. Consider the product of $p_{high}(x)$ and the irreducible polynomial, which may be calculated with carry-free multiplication. The coefficients of x^{d-i} for $0 \leq i < pp_d$ will be equal for both $p(x)$ and this product. Therefore, their sum reduces $p(x)$ by pp_d terms.

Now, suppose that $p(x)$ is the product of $a(x)$ and $b(x)$, which are elements of $GF(2^w)$. The maximum degree of $p(x)$ is $2w - 2$, which means that we have a maximum of $w - 1$ terms to reduce so that the product's degree is less than w. Using carry-free multiplication, we may reduce $p(x)$ by pp_d terms. Therefore, if we perform $\lceil \frac{w-1}{pp_d} \rceil$ such reductions, our result will be an element of $GF(2^w)$.

We illustrate by multiplying $10 = x^3 + x$ and $13 = x^3 + x^2 + 1$ in $GF(2^4)$. Let $p(x)$ be their product. Then:

$$\begin{aligned}
p(x) &= x^6 + x^5 + x^4 + x = (x^2 + x + 1)x^4 + x \\
p_{high}(x) &= x^2 + x + 1 \\
p_{low}(x) &= x
\end{aligned}$$

To reduce $p(x)$ by $pp_d = 3$ terms, we multiply $p_{high}(x)$ by the irreducible polynomial:

$$\begin{aligned}
p_{high}(x)IP(x) &= (x^2 + x + 1)(x^4 + x + 1) \\
&= x^2(x^4 + x + 1) + x(x^4 + x + 1) + (x^4 + x + 1) \\
&= (x^6 + x^3 + x^2) + (x^5 + x^2 + x) + (x^4 + x + 1) \\
&= x^6 + x^5 + x^4 + x^3 + 1
\end{aligned}$$

Adding this to $p(x)$ reduces $p(x)$ by 3 powers of x, and the result is the product in $GF(2^4)$: $x^3 + x + 1 = 11$.

Thus, using the "standard" irreducible polynomials from Figure 5, we may reduce any product in one step for $w = 4$, two steps for $w \in \{8, 64, 128\}$ and four steps for $w \in \{16, 32\}$. If we use the "alternate" irreducible polynomials for $w \in \{16, 32\}$, then we can reduce any product in two steps.

The C code in Figure 8 shows an implementation of *CARRY-FREE* for $GF(2^8)$. It assumes that the procedure **cfm(a,b)** returns the result of a carry-free multiplication of **a** and **b**. The first **cfm()** call creates the polynomial product of **a** and **b**. The bit shifts in the second and third calls create $p_{high}(x)$, which is multiplied by the irreducible polynomial, and then added to the product with exclusive-or. The first of these zeros bits 11 through 14 in the product. The second zeros bits 8 through 10.

The right side of Figure 8 shows a concrete example of multiplying $a = 230$ and $b = 178$ to yield $p = 248$ in $GF(2^8)$. Bits 11 through 14 are highlighted in the first step of the reduction, and bits 8 through 10 are highlighted in the second step.

6.3 The *TABLE* Implementation: Multiplication Tables

Galois field multiplications, like normal multiplications, can be implemented easily using precalculated multiplication tables. When $w \leq 8$, one can store a complete multiplication table in 2^{16} bytes, or 64 KB. Most practical implementations of Reed-Solomon coding for disk systems smaller than 256 drives employ multiplication and division tables

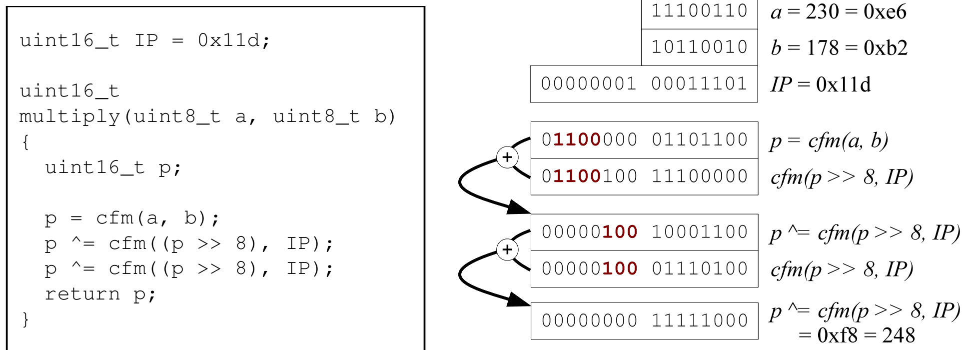


Figure 8: The *CARRY-FREE* implementation in C of **multiply()** in $GF(2^8)$. This assumes that **cfm(a,b)** returns the result of a carry-free multiplication of **a** and **b**.

in this manner [30, 26]. When $w = 16$, a full multiplication table requires 2^{32} 2-byte entries, which requires 8 GB of storage. This is typically too large, so most implementations of Reed-Solomon coding do not use multiplication and division tables for $GF(2^{16})$. However, as detailed in Section 8.3 we can still use multiplication tables for **multiply_region()** for larger values of w if we create a single row of the full multiplication table corresponding to the constant by which the region is being multiplied.

6.4 The *LOG-TABLE* Implementation: Using Discrete Logarithms

There is an additional property of an irreducible polynomial called "primitiveness." If IP is primitive, then each non-zero number in $GF(2^w)$ is equal to x^l for some $0 \leq l < 2^w - 1$. For example, in $GF(2^4)$, $1 = x^0$, $2 = x^1$, $8 = x^3$ and $3 = 8 \times 2 = x^4$. All of the polynomials in Table 5 are primitive as well as irreducible.

The number l is called the *discrete logarithm*. We show the discrete logarithms for all non-zero elements of $GF(2^{2^4})$ below:

Number	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Discrete log	0	1	4	2	8	5	10	3	14	9	7	6	13	11	12

Discrete logarithms may be used to implement multiplication and division efficiently. The mathematics are quite simple, and are analogous to multiplication and division using logarithms in real numbers:

$$\text{If } a = x^l \text{ and b} = x^m\text{, then } ab = x^{l+m}.$$

Thus, we maintain two tables: one of discrete logarithms, and one of inverse logarithms. To calculate the product of a and b, we first compute $r = \log a + \log b$ (using standard addition, not XOR), looking up the discrete logarithms in the table. We then look up r in the inverse logarithm table to find $z = \text{antilog } r$. Using the examples above, to multiply 3 by 4, we add their discrete logarithms (4 + 2 = 6) and take the inverse of the result: $x^6 = 12$.

Because $x^{2^w-1} = 1$, the logarithms cycle every $2^w - 1$ elements. This is important, because the sum of the discrete logarithms can be greater than $2^w - 1$. For example, when we multiply 10 and 13 in $GF(2^{2^4})$, the sum of their logarithms is $9 + 13 = 22$. We can subtract $2^w - 1 = 15$ from the sum to yield the logarithm of the result: $22 - 15 = 7$, and $\log 11 = 7$, so the product of 10 and 13 in $GF(2^4)$ is 11.

```
uint16_t multiply(uint16_t a, uint16_t b)
{
  if (a == 0 || b == 0) return 0;
  return inverse[(uint32_t)log[a] + (uint32_t)log[b]];
}
```

Figure 9: The *LOG-TABLE* implementation in C of **multiply()** in $GF(2^{16})$. The **inverse** table must have $2^{w+1} - 3$ elements. One may instead take the sum modulo $2^w - 1$, and then **inverse** only needs $2^w - 1$ elements.

The simplest C code for *LOG-TABLE* in $GF(2^{16})$ is displayed in Figure 9. However, there are some subtleties in this implementation. First, if the discrete logarithm table holds 16-bit unsigned integers, then the sum of the logs must be a larger data type, as the sum can exceed $2^{16} - 1$. Second, since the sum is a number between 0 and $2(2^w - 2)$, the inverse table must have $2^{w+1} - 3$ entries, of which the last $2^w - 2$ are copies of the first $2^w - 2$. This is because as noted above, the logarithms cycle every $2^w - 1$ elements.

The inverse table may be shrunk by taking the sum modulo $2^w - 1$, trading off decreased space (saving $2^w - 2$ entries) for added time (the modulo instruction).

To create the tables, one starts with $a = 1$, whose discrete logarithm is zero, and one then successively multiplies a by 2 using *SHIFT*, incrementing the discrete logarithm by one, until a equals one again. More detail on this process, including the C code to calculate products and generate the tables, may be found in [22].

To perform division ($\frac{a}{b}$), we simply compute $\log a - \log b = r$ and look up the inverse logarithm for r. Now, the inverse table must have elements for $-(2^w - 2)$ to $2^w - 2$, or one must perform the same modulo operation as in multiplication to reduce the size of the table to $2^w - 1$ entries. If one uses the larger table, then one may use the same table for multiplication and division, so long as it has a total of $2^{w+1} - 2$ elements; The base pointer for the division table needs to be set $2^w - 1$ elements higher than the base pointer for the multiplication table (this is done in Figure 10).

6.4.1 LOG-ZERO: Trading quite a bit of space for one "if" statement

For an even more severe space-time tradeoff, we may eliminate the **if** statement in Figure 9 by defining **log[0]** to have a sentinel value equal to $(2^{w+1} - 2)$. We explain this in two steps. First, suppose we wish to modify the **if** statement in Figure 9 so that it instead reads "**if (b == 0) return 0.**" To do so, we add an extra $2^{w+1} - 2$ elements to the inverse table, whose entries are all zero. To visualize how this works, we show the logarithm and inverse tables for $GF(2^4)$ in Figure 10.

We demonstrate how these tables work with four concrete examples:

- Multiplying 10 and 13. The discrete logarithms are 9 and 13, whose sum is 22. The product may thus be found in element 22 of the inverse table for multiplication, which is 11.
- Dividing 11 by 10. The discrete logarithms are 7 and 9, whose difference is -2. The quotient may thus be found in element -2 of the inverse table for division, which is 13.
- Multiplying 9 and 0. The discrete logarithm for 0 has been set to $(2^{w+1} - 2) = 30$. The sum of the logarithms is thus 44, which is the last element of the inverse table. The product is therefore 0.
- Dividing 0 by 9. Now the difference of the logarithms is 16, and element 16 of the inverse table for division is equal to zero.

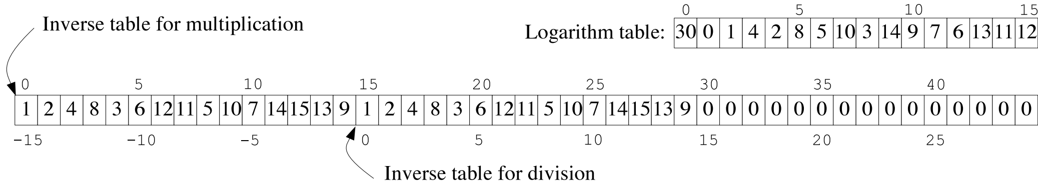


Figure 10: An example of laying out the logarithm and inverse tables using *LOG-ZERO* in $GF(2^4)$. When using these tables, one may eliminate the check for **(a == 0)** in the implementations for both multiplication and division.

An important subtlety is that the values in logarithm table elements must be big enough to represent the sentinel value. For example, when $w = 8$, the sentinel value is bigger than $2^w - 1$, and the logarithm table must hold 16-bit integers rather than eight-bit integers. The inverse table, on the other hand, may still hold 8-bit integers.

We call this optimization *LOG-ZERO*. It has been described previously by Greenan [6] and Luo [15]. It may be extended further to remove the final **if** statement completely by expanding the inverse table to include twice the value of the sentinel. In Figure 10, that would be element 60. In this case, all of the elements from the previous end of the table to this last element are unused, and it is up to the programmer to make use of them for some other purpose. This extension only applies to multiplication and not division, since division by zero is undefined. Additionally, when using this technique for **multiply_region()**, one typically treats multiplication by zero as a special case, and this extension is not necessary.

6.5 The *SPLIT-TABLE* Implementation: Using the Distributive Property to Lower Memory Usage

The distributive property of multiplication applies to Galois Fields:

$$(x + y)z = xz + yz.$$

We leverage this in the *SPLIT-TABLE* implementation as follows. Suppose a is an element of $GF(2^8)$. We may treat a as the sum of two terms, either in its polynomial or binary representation:

Polynomial	Binary
$a(x) = a_{high}(x)x^4 + a_{low}(x)$	$a = (a_{high} << 4) \oplus a_{low}$

In the polynomial representation, a_{high} and a_{low} are polynomials of degree less than four. In the binary representation, they are four-bit numbers. We multiply a and b by splitting the eight bits of a into its two four-bit components a_{high} and a_{low}, and then using them to look up $(a_{high}x^4)b$ and $a_{low}b$ in two different lookup tables. The two products are added to yield the result.

For example, let's return to multiplying 230 and 178 in $GF(2^8)$. In binary, 230 equals 11100110. Therefore, $230 = 11100000 \oplus 00000110 = (1110 << 4) \oplus 0110$. We may use two lookup tables to calculate the product 248:

$$\begin{aligned} 00000110 \times 178 &= 139(10001011) \\ 11100000 \times 178 &= 115(01110011) \\ 139 \oplus 115 &= 248(11111000) \end{aligned}$$

The advantage of this approach is that the lookup tables are smaller than for the *TABLE* implementation. For example, instead of requiring one 256×256 table, multiplication in $GF(2^8)$ with *SPLIT-TABLE* requires two 16×256 tables.

To define *SPLIT-TABLE* generally, we use the polynomial representation of numbers. Let $a_i^g(x)$ be a polynomial of degree less than g. Then a polynomial $a(x) \in GF(2^w)$ may be defined as a sum of these polynomials:

$$a(x) = \sum_{i=0}^{\lceil \frac{w}{g} \rceil - 1} a_i^g(x) x^{gi}. \quad (3)$$

For *SPLIT-TABLE*, we choose two values of g — one for a and one for b — and call them g_a and g_b. We define a and b as sums of smaller polynomials, and define their product as sums of the products of the smaller polynomials. Formally:

$$\begin{aligned} ab = a(x)b(x) &= \left(\sum_{i=0}^{\frac{w}{g_a}-1} a_i^{g_a}(x) x^{g_a i} \right) \left(\sum_{j=0}^{\frac{w}{g_b}-1} b_j^{g_b}(x) x^{g_b j} \right) \\ &= \sum_{i=0}^{\frac{w}{g_a}-1} \sum_{j=0}^{\frac{w}{g_b}-1} a_i^{g_a}(x) b_j^{g_b}(x) x^{g_a i + g_b j} \end{aligned}$$

In the example above, $g_a = 4$ and $g_b = 8$, and therefore the product is the sum of two sub-products. Consider a second, more complex example, where $g_a = g_b = 8$, and we multiply a and b in $GF(2^{32})$. Then:

$$ab = a(x)b(x) = \sum_{i=0}^{3} \sum_{j=0}^{3} a_i^8(x) b_j^8(x) x^{8i+8j}$$

Multiplying out, the product is a sum of 16 terms:

$$\begin{aligned} ab = \; & a_0^8(x)b_0^8(x) + a_0^8(x)b_1^8(x)x^8 + a_0^8(x)b_2^8(x)x^{16} + a_0^8(x)b_3^8(x)x^{24} + \\ & a_1^8(x)b_0^8(x)x^8 + a_1^8(x)b_1^8(x)x^{16} + a_1^8(x)b_2^8(x)x^{24} + a_1^8(x)b_3^8(x)x^{32} + \\ & a_2^8(x)b_0^8(x)x^{16} + a_2^8(x)b_1^8(x)x^{24} + a_2^8(x)b_2^8(x)x^{32} + a_2^8(x)b_3^8(x)x^{40} + \\ & a_3^8(x)b_0^8(x)x^{24} + a_3^8(x)b_1^8(x)x^{32} + a_3^8(x)b_2^8(x)x^{40} + a_3^8(x)b_3^8(x)x^{48}. \end{aligned}$$

Each $a_i^8(x)$ and $b_j^8(x)$ is an 8-bit number, so each product may be calculated with a 256×256 element table. Since each product is an element of $GF(2^{32})$, these tables hold 32-bit words. Only one table is needed for each value of x^{8i}, which means a total of seven tables, or $7(4)(256)(256) = 1.75$ MB. One may similarly use $g_a = g_b = 8$ to implement multiplication in $GF(2^{64})$ by computing 64 sums of terms that may be looked up in 15 tables of $2^8 \times 2^8$ elements. The total memory requirement for this approach is just 7.5 MB.

SPLIT-TABLE is an important technique for a variety of reasons. Without a carry-free instruction like `pclmulqdq`, the version above with $g_a = g_b = 8$ is the fastest way to perform **multiply()** in $GF(2^{32})$. This is the implementation used in version 1.2A of the **jerasure** erasure-coding library [26]. When $g_a = 4$ and $g_b = w$, *SPLIT-TABLE* may be employed to leverage the `pshufb` instruction to perform **multiply_region()** at cache line speeds (see section 8.4), as first reported by Li and Huan-yan [13] and then leveraged by Anvin to implement RAID-6 decoding in the Linux kernel [1].

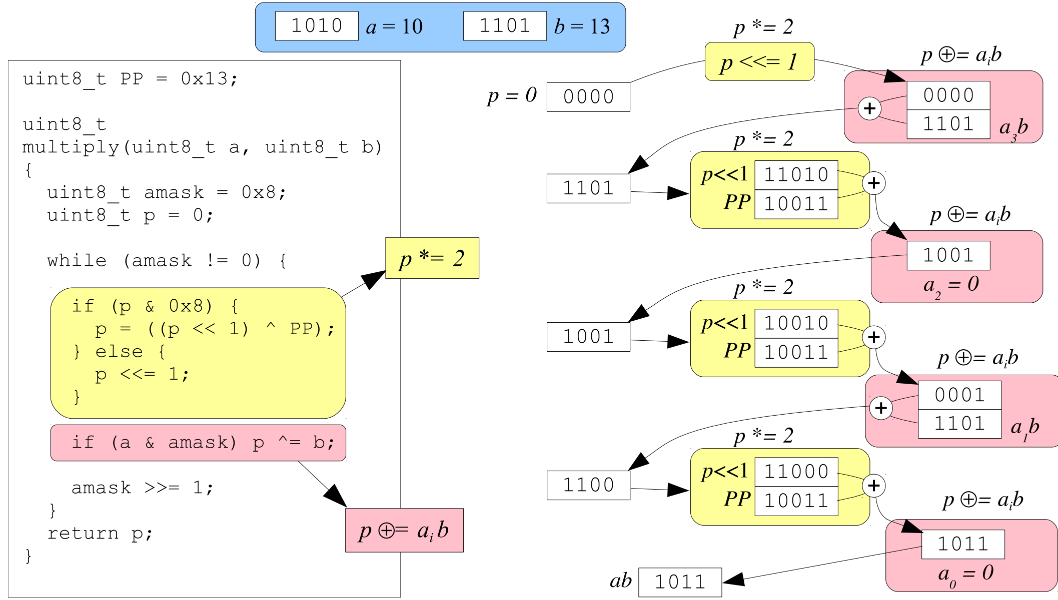


Figure 11: The $BYTWO_p$ implementation in $GF(2^4)$. The left side is an implementation in C. The right side demonstrates how $a = 10$ (1010) times $b = 13$ (1101) equals = 11 (1011).

6.6 The $BYTWO_p$ and $BYTWO_b$ Implementations: Incrementally reducing the product.

The *BYTWO* implementations are best motivated with an example. Let a and b be numbers in $GF(2^4)$. The polynomial representation of a is $a_3x^3 + a_2x^2 + a_1x + a_0$, where each a_i is a zero or one. Then the *SHIFT* implementation calculates ab by evaluating the following equation:

$$ab = \mathbf{reduce}(((a_3b)x^3 + (a_2b)x^2 + (a_1b)x + a_0b).$$

Thus, **reduce()** must work on a polynomial of degree up to six, which requires more than w bits to store. The $BYTWO_p$ implementation computes the product incrementally, calling **reduce()** at each step. It is defined in Equation 4.

$$ab = a_0b + \mathbf{reduce}(x(a_1b + \mathbf{reduce}(x(a_2b + \mathbf{reduce}(xa_3b))))). \tag{4}$$

By calling **reduce()** each time we multiply by x, we never have to use numbers that are greater than w bits. We call this $BYTWO_p$ because the polynomial x is equal to the decimal number 2, and thus we are really incrementally multiplying the product by two. To hammer this point home, equation 5 shows the decimal representation of $BYTWO_p$:

$$ab = a_0b \oplus 2(a_1b \oplus 2(a_2b \oplus 2(a_3b))) \tag{5}$$

The left side of Figure 11 shows a C implementation of $BYTWO_p$ in $GF(2^w)$, and the right side demonstrates how $10 * 13 = 11$ is calculated with this implementation. Obviously, for fixed values of w, one may unroll the **while** loop completely, thereby improving performance further.

```
uint8_t PP = 0x13;

uint8_t
multiply(uint8_t a, uint8_t b)
{
  uint8_t p = 0;

  while (1) {
    if (a & 1) p ^= b;

    a >>= 1;
    if (a == 0) return p;

    if (b & 0x8) {
      b = ((b << 1) ^ PP);
    } else {
      b <<= 1;
    }
  }
  return p;
}
```

Figure 12: The *BYTWO*$_b$ implementation in $GF(2^4)$. The left side is an implementation in C. The right side demonstrates how $a = 10$ (1010) times $b = 13$ (1101) equals 11 (1011).

Instead of incrementally multiplying the product by two, we can instead multiply b by two incrementally. This is the *BYTWO*$_b$ implementation. It is based on equations 6 (polynomial) and 7 (decimal).

$$ab = a_0 b + a_1(\mathbf{reduce}(bx)) + a_2(\mathbf{reduce}(bx^2)) + a_3(\mathbf{reduce}(bx^3)). \quad (6)$$

$$ab = a_0 b \oplus a_1(2b) \oplus a_2(4b) \oplus a_3(8b). \quad (7)$$

As with *BYTWO*$_p$, we show an implementation of *BYTWO*$_b$ in C and demonstrate how it calculates $10 * 13 = 11$ in $GF(2^4)$. An important difference between *BYTWO*$_p$ and *BYTWO*$_b$ is that the loop in *BYTWO*$_b$ terminates after it sees the highest one bit in a. Therefore, the loop only executes once when a equals two, regardless of the value of w. This implementation was leveraged by Anvin to implement RAID-6 encoding in the Linux kernel, where values are successively multiplied by two very quickly [1]. Please see Section 8.5 for how Anvin also multiplies multiple words by two in just a few machine instructions.

6.7 The *GROUP* Implementation: Adding table lookups to *SHIFT*

While the implementation of *SHIFT* is simple, it can be very slow, particularly for large fields, requiring w iterations for multiplication in $GF(2^w)$. The number of iterations can be sped up by shifting multiple bits at a time, and using a small table lookup to compute each product; we call this the *GROUP* implementation. By doing so, reduction becomes a bit more complex as well, so, as with previous implementations, we illustrate the *GROUP* implementation with an example. Suppose we are are again multiplying $a = 230$ and $b = 178$ in $GF(2^8)$. Instead of viewing a as eight bits, we may view it as four two-bit numbers:

$$230 = (11100110) = |11|10|01|10| = |3|2|1|2|$$

We then create a four-element table, M, which holds the product of b and zero through three, and we use that table to calculate the carry-free multiplication with three shifts and XORs. The process is illustrated in Figure 13. By using

the multiplication table to aggregate SHIFT operations, we perform the carry-free multiplication two bits at a time, rather than one.

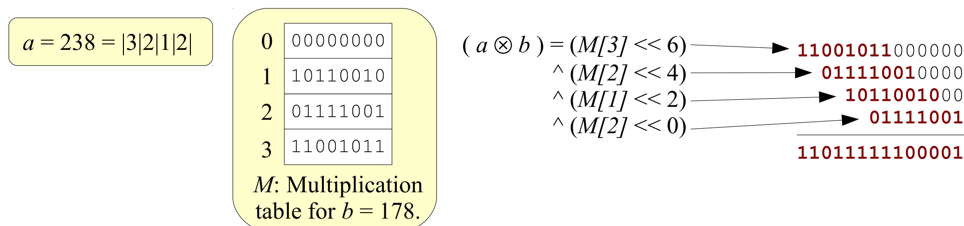


Figure 13: Using a four-element multiplication table to calculate $a \otimes b$ when $a = 230$ and $b = 178$.

We may also create a four-element table, R, from the irreducible polynomial, which allows us to reduce the product back to eight bits by operating on two bits at a time, rather than one, as illustrated in Figure 13, where we show how $230 \otimes 178$ reduces to 248.

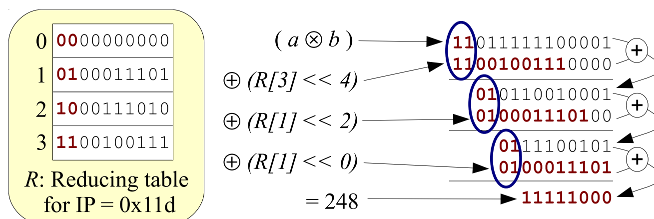


Figure 14: Using a four-element reducing table to reduce $a \otimes b$ when $a = 230$ and $b = 178$.

Extrapolating from this example, we define the *GROUP* optimization using two parameters, g_m and g_r. These are the numbers of bits in the indices of the multiplication and reducing tables respectively. In our example above, they both equal two. There are tradeoffs in the size of these values. Bigger values of g_m and g_r allow us to perform carry-free multiplication and reduction in fewer steps, but are of size 2^{g_m} and 2^{g_r} respectively. Moreover, the multiplication table needs to be created anew for each value of b, which argues for smaller values of g_m. The reducing table, on the other hand, is created from the irreducible polynomial, and therefore only needs to be created once for all values of b. Thus, it makes sense for g_r to be bigger than g_m.

We remark that when creating the R table, it is not always the case that the element in index i is equal to $i \otimes IP$, as it is in Figure 14. Instead, the element in index i needs to be equal to $x \otimes IP$ for whatever value of x makes the first g_m bits of the product equal i. For example, suppose we use the irreducible polynomial $x^4 + x^3 + 1$ to implement $GF(2^4)$, and we set $g_m = 2$. Then, $2 \otimes IP$ equals $x^5 + x^4 + x$, or 110010 in binary. Thus, 110010 goes into index 3 of R, and not into index 2.

As with *SHIFT*, the carry-free multiplication step of *GROUP* creates a word that is longer than w bits. When $g_m = g_r$, we may alternate multiplication and reduction so that we only deal with w-bit words. The process is a bit subtle, so we give a detailed example in Figure 15, which again multiplies 230 by 178 in $GF(2^8)$ when $g_m = g_r = 2$. We split a into two-bit indices, $|3|2|1|2|$, and build M from b as before. We assume that R has been created already; however it only holds the smallest eight bits, rather than ten bits as in Figure 14.

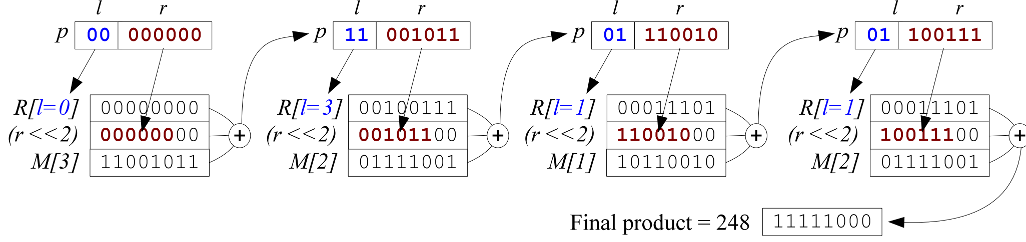


Figure 15: When $g_m = g_r$, we can alternate multiplying and reducing so that we only have to use w-bit words in the *GROUP* calculation. In this example, we multiply 230 and 178 in $GF(2^4)$.

To perform the calculation, we incrementally build a product p, starting with the value zero. We split p into two parts, l and r, which are two and six bits respectively. We then perform each step of multiplication as follows. We shift r two bits to the left, XOR it with $M[3]$ and with $R[l]$ to yield the next value of p. We split that into l and r and repeat the process with the next word of a. In other words, we shift r two bits to the left, and XOR it with $M[2]$ and $R[l]$ to yield the next value of p. We perform this step two more times, using $M[1]$ and finally $M[2]$, to yield the final product.

We may perform similar optimizations when g_r is a multiple of g_m. For example, if $g_r = 2g_m$, then we perform two multiplications and one reduction at each iteration.

GROUP was used by Luo *et al.* to perform multiplications of large words $w \in \{32, 64\}$ in HAIL [15]. We note that *BYTWO*$_p$ is in fact equivalent to *GROUP* with $g_m = g_r = 1$.

6.8 The *COMPOSITE* Implementation: A Different Kind of Field

Composite Galois Fields [6, 15] are denoted $GF((2^l)^k)$, and implement finite fields for $w = lk$. Unlike the other implementations in this paper, composite Galois Fields implement a different finite field. In other words, $GF((2^l)^k) \neq GF(2^{lk})$. However, they have the appropriate properties, and can thus be used in applications that require finite fields.

As with $GF(2^w)$, numbers in $GF((2^l)^k)$ have four representations: decimal, hexadecimal, binary and polynomial. However, instead of a $w-1$-degree polynomial with binary coefficients, the polynomial is a $k-1$ degree polynomial whose coefficients are elements of $GF(2^l)$. For example, the numbers 230 and 178 in $GF((2^4)^2)$ would have the following polynomial representations.

$$
\begin{aligned}
230 &= |1110|0110| = \quad 14x + 6 \\
178 &= |1011|0010| = \quad 11x + 2
\end{aligned}
$$

Addition and multiplication are defined similarly to $GF(2^w)$. As before, addition is equal to XOR. Multiplication is standard polynomial multiplication, where the arithmetic for the coefficients is in $GF(2^l)$, and the product is taken modulo a irreducible polynomial of degree k. Irreducible polynomials for many w, k, l combinations may be found in [15].

When we restrict our attention to $k = 2$, and $GF(2^l)$ is implemented with the "standard" irreducible polynomials from Figure 5, the following are irreducible polynomials for $GF((2^l)^2)$:

$$
\begin{aligned}
GF((2^4)^2): &\quad x^2+2x+1\\
GF((2^8)^2): &\quad x^2+3x+1\\
GF((2^{16})^2): &\quad x^2+2x+1\\
GF((2^{32})^2): &\quad x^2+2x+1
\end{aligned}
$$

To demonstrate multiplication, we multiply 230 and 178 in $GF((2^4)^2)$:

$$
\begin{aligned}
(230)(178) &= (14x+6)(11x+2)\\
&= (14*11)x^2+(14*x+11*6)x+(6*2)\\
&= 8x^2+(15+15)x+12\\
&= 8x^2+12
\end{aligned}
$$

To reduce this, we multiply the irreducible polynomial by eight and add it to the product:

$$
\begin{aligned}
&= 8x^2+12+8(x^2+2x+1)\\
&= 8x^2+12+8x^2+3x+8\\
&= 3x+4\\
&= |0011|0100| = 0x34 = 52.
\end{aligned}
$$

When the irreducible polynomial is of the form x^2+sx+1, we may generalize multiplication as follows:

$$
\begin{aligned}
p &= ab\\
p_1x+p_0 &= (a_1x+a_0)(b_1x+b_0)\\
&= a_1b_1x^2+a_1b_0x+a_0b_1x+a_0b_0
\end{aligned}
$$

We take this modulo the irreducible polynomial by adding a_1b_1 times the irreducible polynomial:

$$
\begin{aligned}
p_1x+p_0 &= a_1b_1x^2+a_1b_0x+a_0b_1x+a_0b_0+a_1b_1(x^2+sx+1)\\
&= (a_1b_0+a_0b_1+a_1b_1s)x+(a_0b_0+a_1b_1)
\end{aligned}
$$

Therefore:

$$
p_1 = a_1b_0+a_0b_1+a_1b_1s \tag{8}
$$

$$
p_0 = a_0b_0+a_1b_1 \tag{9}
$$

This allows us to implement multiplication in $GF((2^l)^2)$ with five multiplications in $GF(2^l)$ and three XOR operations. For large field sizes, this may be the most efficient way to perform multiplication. It also has interesting implications for implementing **multiply_region()**, as discussed in Section 8.6.3 below.

6.9 Using Bit Matrices

In the specification of Cauchy Reed-Solomon coding, Blömer represents an element a in $GF(2^w)$ with a $w \times w$ bit-matrix [2]. Each column i ($0 \leq i < w$) contains the binary representation of ax^i. Then, the product of the matrix representations of a and b is equal to the matrix representation of ab. Additionally, if one turns a number's binary representation into a column vector, the product of a's matrix with b's vector yields the column vector of ab. Functionally, this is equivalent to *BYTWO*$_b$. Figure 16 illustrates both types of multiplication for the values 10 and 13 in $GF(2^4)$. By convention, the numbers are mapped to column vectors with low bits above high bits. In that way, the matrix representation of one is an identity matrix.

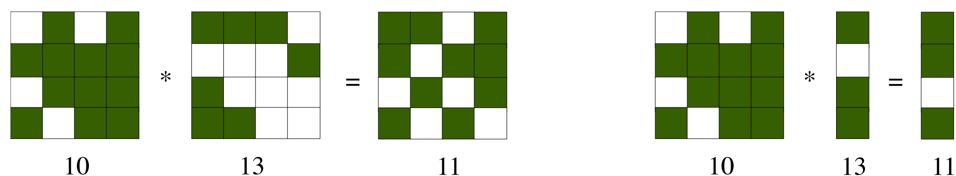


Figure 16: Matrix-matrix and matrix-vector multiplication to implement multiplication in $GF(2^4)$.

Because this is functionally equivalent to *BYTWO*$_b$, we do not give it a name or include it in the summary below. However, employing bit matrices can facilitate division (see section 7.1), or fast region operations, because only XOR is required (see section 8.6.1).

6.10 Single Multiplication Summary

Figure 17 summarizes the implementation techniques we have described for **multiply()**.

7 Division

When w is small, division is most easily accomplished by creating a division table at the same time as the multiplication table in section 6.3. As detailed in Section 6.4, division in *LOG-TABLE* is identical to multiplication, except the logarithms are subtracted rather than added. When w is too large to use *TABLE* or *LOG-TABLE*, we have to resort to other, more expensive techniques that calculate the inverse of a number. Then, to divide a by b, we multiply a by b's inverse.

7.1 Inverse using bit matrices

An convenient feature of the bit matrix representation of $GF(2^w)$ is that inverting a number's bit matrix yields the bit matrix of the number's inverse. When a field is too large to store logarithm tables, this method of finding a number's inverse becomes a practical alternative.

7.2 Euclid's Algorithm

A different way to find an element's inverse is to employ the extended version of Euclid's algorithm [4], which finds the inverse of an element b using multiplication and addition. Euclid's algorithm proceeds iteratively and uses the

Technique	Running Time Complexity	Space Overhead
SHIFT	$O(w)$	$O(1)$
CARRY-FREE	$O(1)$	$O(1)$
TABLE	$O(1)$	$O(2^{2w})$
LOG-TABLE	$O(1)$	$O(2^w)$ to $O(2^{w+2})$
SPLIT-TABLE	$O\left(\frac{w}{g_a} \times \frac{w}{g_b}\right)$	$O\left((g_a + g_b)2^{(g_a g_b)}\right)$
GROUP	$O\left(\frac{w}{g_m} + \frac{w}{g_r} + 2^{g_m}\right)$	$O(2^{g_m} + 2^{g_r})$
BYTWO$_p$	$O(w)$	$O(1)$
BYTWO$_b$	$O(1)$ to $O(w)$, depending on a	$O(1)$
COMPOSITE	O(Complexity of the base field)	Overhead of the base field.

Figure 17: Summary of the running time complexity and space requirements for different Galois field multiplication techniques.

polynomial representation of field elements. The goal is to find a sequence of elements $E_0, E_1, \ldots$ such that the degree of E_{i+1} is smaller than the degree of E_i, and $E_{i+1} = E_{i-1} - c_i E_i$ for some element c_i. We start with $E_0 = IP$ and $E_1 = b$, and we continue until $E_i = 1$. This is guaranteed to work because of the "irreducible" property of the irreducible polynomial [21].

We use the E_i and c_i to calculate the inverse in the following way. We may write each E_i as a linear combination of E_0 and E_1: $E_i = y_i E_0 + z_i E_1$. Clearly, $y_0 = 1$, $z_0 = 0$, $y_1 = 0$ and $z_1 = 1$. We use the fact that $E_{i+1} = E_{i-1} - c_i E_i$ to calculate y_{i+1} and z_{i+1}:

$$
\begin{aligned}
y_{i+1} &= y_{i-1} - c_i y_i \\
z_{i+1} &= z_{i-1} - c_i z_i
\end{aligned}
$$

When we're done, we've reached a point where $E_i = 1$. Therefore:

$$
\begin{aligned}
1 &= y_i E_0 + z_i E_1 \\
&= y_i IP + z_i b \\
&= \textbf{reduce}(y_i IP) + \textbf{reduce}(z_i b) \\
&= 0 + \textbf{reduce}(z_i b)
\end{aligned}
$$

Therefore, z_i is the inverse of b. All that remains is to find the c_i, which can be done with the following iterative steps:

- Set $c_i = 0$.
- While the degree of $E_{i-1} - c_i E_i$ is greater than or equal to the degree of E_i, do the following:
- Let d be the degree of $E_{i-1} - c_i E_i$ and d' be the degree of E_i.
- Set $c_i = c_{i-1} + x^{d-d'}$.

i	E_i	y_i	z_i	c_i	$E_{i+1} = E_{i-1} - c_i E_i$
0	$x^4 + x + 1$	1	0	–	–
1	$x^3 + x^2 + 1$	0	1	$x + 1$	x^2
2	x^2	1	$x + 1$	$x + 1$	1
3	1	$x + 1$	$(x + 1)(x + 1) - 1 = x^2$	–	–

Figure 18: Using Euclid's algorithm to calculate the inverse of 13 in $GF(2^4)$.

For example, suppose $E_{i-1} = x^4 + x + 1$ and $E_i = x^3 + x^2 + 1$. The steps above calculate $c_i = x + 1$, and $E_{i+1} = x^2$. We use this calculation for a concrete example in Figure 18. In this example, we calculate the inverse of $13 = x^3 + x^2 + 1$ in $GF(2^4)$. Since $E_3 = 1$, the inverse of 13 is equal to $z_3 = x^2 = 4$.

The running time of Euclid's algorithm is $O(w)$. With composite fields, Euclid's algorithm is a little more complex, since coefficients of the x^i terms may be larger than one. However, it is a straightforward modification that employs division in the base field to calculate each c_i.

8 Multiplying a Region by a Constant

The most costly operation in erasure coding applications is multiplying a large region of bytes by a constant in $GF(2^w)$. There are considerations and tricks that can be applied to make multiplying a region by a constant much faster than performing single multiplications for every word in the region. Some of these concern memory; some reduce instruction count, and some employ operations on large words to operate on multiple smaller words simultaneously. We detail them all below, going from simplest to most complex. Each implements **multiply_region(uint$_w$_t** a**, void** *b**, void** *p**, int** $bytes$**, bool** add**)**.

8.1 Only One Table Lookup

The first optimization is obvious. In implementations that require table lookups, one only needs to look up the values for a once, rather than for every multiplication. This reduces the instruction count of calling **multiply()** for every w-bit word in b.

8.2 Double-Tables and Quad-Tables

Consider $w = 4$. The multiplication table for this value of w is a 16×16 table of 4-bit quantities, which works well for implementing **multiply()**. With **multiply_region()**, one may consider each group of 8 bits in b to be two 4-bit words and employ a single 16×256 table to look up the product of a with both words simultaneously. We call this a *Double-Table*, whose size is $\frac{2w}{8} \times 2^w * 2^{2w}$ bytes. The Double-Table for $w = 4$ uses 4 KB; for $w = 8$, it occupies 32 MB.

For $w = 4$, one may even employ a *Quad-Table*, which operates on two bytes at a time by multiplying four 4-bit words by a simultaneously, since the size of this table is only 2 MB.

8.3 Lazy Table Creation

In the *TABLE* implementation, a single call to **multiply_region()** does not use the entire multiplication table, but instead only uses the row of the table that corresponds to a. Rather than storing the entire multiplication table in memory, a

lazy implementation of *TABLE* creates the proper row of the table at the beginning of each call to **multiply_region()**. If the size of the region is large enough, the cost of creating the row of the table is amortized, and the lazy *TABLE* implementation may outperform other implementations. It may even outperform a non-lazy *TABLE* implementation due to cache considerations.

For example, when $w = 16$, a full multiplication table requires 2^{32} 16-bit words, occupying 8 GB of memory, which is prohibitively large. However, a single row of the table may be stored in 2^{16} 16-bit words, or just 128 KB. Using this table, the product of every two bytes of b may be calculated with one table lookup, as opposed to two lookups and some arithmetic when employing *LOG-TABLE*. Similarly, one may implement a lazy Double-Table in $GF(2^8)$ or a lazy Quad-Table in $GF(2^4)$ in the same 128 KB.

Finally, we may incorporate laziness very effectively with the *SPLIT-TABLE* implementations when $g_b = w$. At the beginning of the **multiply_region()** operation, we must create $\frac{w}{g_a}$ tables, each of which contains 2^{g_a} elements of $GF(2^w)$. Then, each word of the product region is calculated with $\frac{w}{g_a}$ table lookups and XORs.

For example, a lazy implementation of *SPLIT-TABLE* in $GF(2^{32})$ with $g_a = 8$ and $g_b = 32$ requires four 256-element tables which must be created at the beginning of the **multiply_region()** operation. Then, every 32-bit word in the product region may be calculated with four table lookups, and either three or four XOR's (depending on the value of *add*). We will see below in Sections 8.4 and 8.6.2 that setting g_a to four can yield additional improvements due to vector instructions.

Finally, with the *GROUP* implementation, the multiplication table may be calculated once at the beginning of each **multiply_region()** operation, and thus g_m may be bigger than it is for **multiply()**.

8.4 Small Table Lookup with Vector Instructions

Intel's Streaming SIMD Instructions [10] have has become ubiquitous in today's commodity microprocessors. They are supported in CPUs sold by Intel, AMD, Transmeta and VIA; the ARM instruction set supports similar instructions. Although differences in the instruction set prevent the use of Intel code directly on ARM chips, techniques described here can be applied to ARM processors as well. 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 can leverage the following instructions to optimize the performance of **multiply_region()**:

- **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.
- **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)** (also known as `pshufb`) 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.

This last instruction has a profound impact on implementing **multiply_region()**, because it allows us to perform 16 simultaneous table lookups of a 16-element table of bytes. We give a very concrete example in Figure 19. Here we want to multiply a region b of 16 bytes by the number 7 in $GF(2^4)$. We create a 16 element table, *table1*, which is a multiplication table for the number 7, and then a second table *table2*, which is identical to *table1*, except all the entries are shifted four bits to the left. Additionally, we have a bit mask, *mask1*, which isolates the rightmost four bits

in each byte, and a second mask *mask2* which isolates the leftmost four bits. All elements in Figure 19 are shown in hexadecimal.

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

Figure 19: Using SIMD instructions to perform **multiply_region()** of a 128-bit region b by 7 in $GF(2^4)$. The 128-bit vectors are denoted as 16 two-digit numbers in hexadecimal.

The first instruction performs a 128-bit AND operation on b and *mask1*, which isolates the rightmost four bits of each byte. The second instruction performs the 16 simultaneous table lookups, to multiply each of the rightmost four bits by seven in $GF(2^4)$. The third through fifth instructions isolate the leftmost four bits, shift them right by four bits and then use them to perform 16 simultaneous table lookups to multiply them by seven in $GF(2^4)$. The final instruction combines the two products and completes the operation.

Thus, after setting up the tables and masks, we may perform 16 bytes worth of multiplications in six vector operations, which is a drastic improvement over the previous *TABLE* implementation.

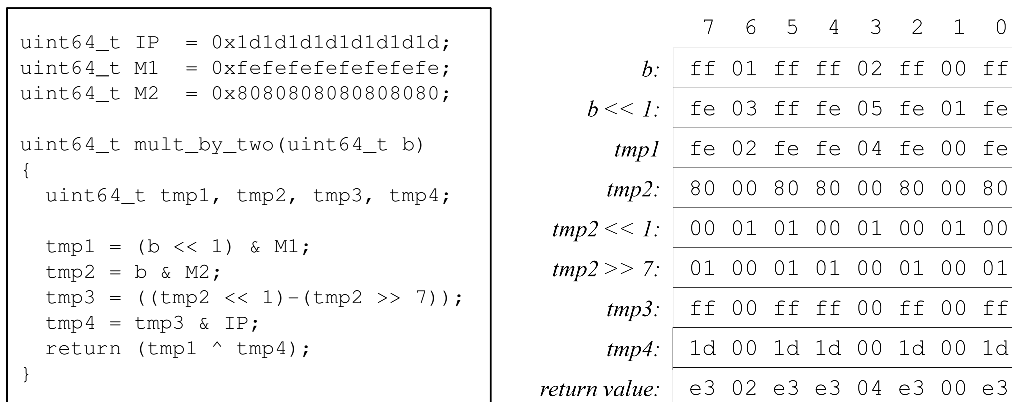
We may implement *SPLIT-TABLE* in $GF(2^8)$ in an almost identical fashion. We set g_a to 4 and g_b to 8, which means that multiplication requires two 16-byte tables – one for the high four bits of each byte and one for the low four bits. These two tables are used in place of *table1* and *table2* in Figure 19.

For larger w, we may leverage **mm_shuffle_epi8()** similarly, but some difficulties arise because the vectors may only represent 16-element tables of bytes, and not tables of larger elements. We discuss how to use memory layout to leverage these instructions below in section 8.6.2.

8.5 Anvin's Optimization for multiplying by two in parallel

For the restricted case of multiplying a region of elements of $GF(2^w)$ by the number two, Anvin unearthed a brilliant optimization [1]. We illustrate in Figure 20, which shows the C code and a concrete example of multiplying a 64-bit integer b, that holds eight numbers in $GF(2^8)$, by two.

The variable *IP* holds eight copies of the irreducible polynomial for $GF(2^8)$, minus the x^8 terms. *M1* creates the variable *tmp1*, which holds all eight numbers in b, shifted one bit to the left. *M2* isolates the highest bits of each byte of b, which are stored in *tmp2*. *tmp3* creates a bitmask from *tmp2* in the following way: if the high bit of a byte is set in *tmp2*, then all of the bits for that byte are set in *tmp3*. *tmp3* is then used to create *tmp4*, where each byte is equal to the irreducible polynomial only if its bit in *tmp2* is set. Therefore, the irreducible polynomial is only applied to the bytes that need it. The final eight products are created by performing the XOR of *tmp1* and *tmp4*.



```
uint64_t IP  = 0x1d1d1d1d1d1d1d1d;
uint64_t M1  = 0xfefefefefefefefe;
uint64_t M2  = 0x8080808080808080;

uint64_t mult_by_two(uint64_t b)
{
  uint64_t tmp1, tmp2, tmp3, tmp4;

  tmp1 = (b << 1) & M1;
  tmp2 = b & M2;
  tmp3 = ((tmp2 << 1)-(tmp2 >> 7));
  tmp4 = tmp3 & IP;
  return (tmp1 ^ tmp4);
}
```

	7	6	5	4	3	2	1	0
b:	ff	01	ff	ff	02	ff	00	ff
b << 1:	fe	03	ff	fe	05	fe	01	fe
tmp1	fe	02	fe	fe	04	fe	00	fe
tmp2:	80	00	80	80	00	80	00	80
tmp2 << 1:	00	01	01	00	01	00	01	00
tmp2 >> 7:	01	00	01	01	00	01	00	01
tmp3:	ff	00	ff	ff	00	ff	00	ff
tmp4:	1d	00	1d	1d	00	1d	00	1d
return value:	e3	02	e3	e3	04	e3	00	e3

Figure 20: Multiplying eight numbers by two in $GF(2^8)$ simultaneously using Anvin's optimization.

In the concrete example, we multiply 8 bytes, of which five are 0xff and the remainder are 0, 1 and 2, by two. The example illustrates how *tmp4* only contains the irreducible polynomial for the bytes whose highest bit is equal to one (the 0xff bytes).

Anvin's optimization is employed in Linux's implementation of RAID-6. Since the central operation in both *BYTWO* implementations is to multiply either p or b by two, Anvin's optimization may be employed to implement **multiply_region()** by working on regions of 64 bits (using 8-byte integers) or 128 bits (using SIMD instructions) at a time.

8.6 Alternate mappings of words to memory

The natural way to store elements in a Galois Field is to partition memory so that each sequence of w consecutive bits is an element of $GF(2^w)$. All of the descriptions to this point have assumed this way of storing the elements. However, there are some improvements to region operations that may be achieved by splitting each element over multiple regions.

To be precise, suppose that we have n words in $GF(2^w)$ that we wish to store in a region of memory, R. We label the n words $a_0, \ldots, a_{n-1}$, and we label the individual bits of each a_i as $a_{i,0}, \ldots, a_{i,w-1}$. R is composed of nw bits, which we label $r_0, \ldots, r_{nw-1}$. The standard way to store the words in R is consecutively: Bit $a_{i,j}$ is stored in r_{wi+j}.

However, we may define alternate ways to store the bits. We call these *alternate mappings*, and they are parameterized by integers x and y such that $w = xy$. Then we may partition R into x subregions, $R_0, \ldots, R_{x-1}$ so that subregion R_0 stores bits $a_{i,0}$ through $a_{i,y-1}$ for all i, subregion R_1 stores bits $a_{i,y}$ through $a_{i,2y-1}$ for all i, and so on. To be precise, bit $a_{i,j}$ is stored in $r_{(j/y)*yn+iy+j\%y}$, where the first term uses integer division.

In Figure 21, we give three example mappings of four words in $GF(2^4)$. To match our previous examples, we order the bits in descending order from left to right. The first example has $x = 1$ and $y = 4$, and the four words are stored in consecutive bits. This is the standard mapping. In the second example, there are two subregions: R_0, which stores the first two bits of each word, and R_1, which stores the second two bits. The final example has four subregions, where each subregion R_i stores bit i of each word.

In the following sub-sections, we show how various implementations of **multiply_region()** may be improved with an alternate mapping. It is important to note that unless backward compatability is required for an application, there

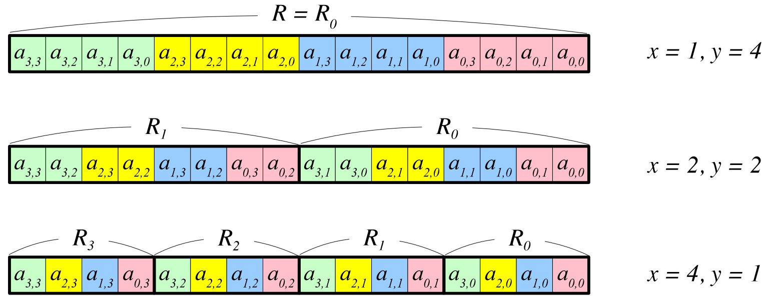


Figure 21: Three examples of storing four words in $GF(2^w)$ in 16 bits of memory.

is no need to "convert" from a standard mapping to an alternate mapping. As long as the same mapping is employed consistently by an erasure coding application, there is no need to worry about how words are mapped to memory. The important property is that addition and multiplication of a region by a constant have the correct mathematical properties. In other words, if a region b is multiplied by a and then multiplied by $\frac{1}{a}$, the resulting region should equal the original b, regardless of what mapping is employed.

8.6.1 Cauchy Reed-Solomon Coding

In Cauchy Reed-Solomon Coding, the mapping chosen is $x = w$ and $y = 1$ [2]. Thus, there are w subregions, $R_0, \ldots, R_{w-1}$, and subregion R_i holds the i-th bit of each word. To implement **multiply_region(**a**,** b**,** p**,** $\frac{wn}{8}$**,** add**)** one employs the matrix representation of a and vector representation of b described in Section 6.9. The mapping allows one to multiply multiple words of b incrementally, employing only the XOR operation.

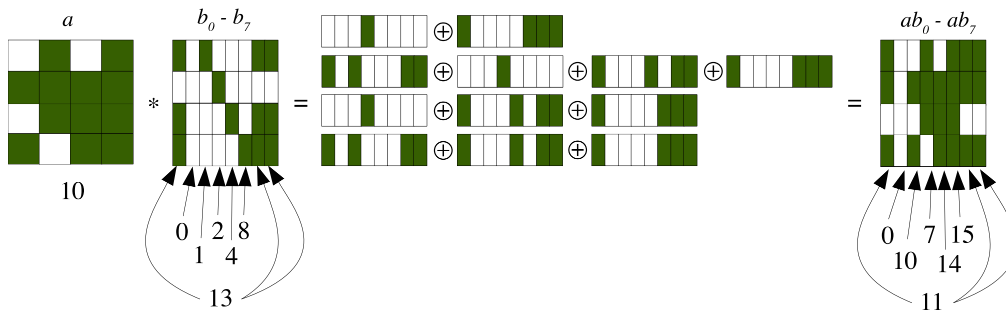


Figure 22: Employing the matrix/vector representation and a mapping where $x = w$ and $y = 1$ to multiply eight words of b by $a = 10$ with XOR operations on bytes.

We draw an example in Figure 22. Here, we multiply b_0 through b_7 by $a = 10$ in $GF(2^4)$. We represent a with a matrix and the eight words of b as eight column vectors. This employs the mapping where $x = 4$ and $y = 1$. Each row is a byte, and therefore the product matrix may be calculated by performing XORs on bytes. For example, the first row of the product matrix is calculated as the XOR of the second and fourth rows of b. The second row is the

XOR of all four rows. Thus, the eight products are calculated with a total of eight XORs on single bytes. Practical implementations of this mapping make sure that each subregion is at least 128 bits, so that large XOR operations may be performed.

In Cauchy Reed-Solomon coding, the subregions are called "packets," and their size is arbitrary. In practical implementations of Cauchy Reed-Solomon coding, the size of the subregions and the ordering of the XOR operations have a significant, but hard-to-quantify relationship on the performance of the multiplication, mainly because of the caches [17, 24]. Moreover, one may reduce the number of XOR operations by leveraging intermediate expressions [7, 8, 25]. This implementation of Galois Field arithmetic has been at the heart of the erasure coding engines of Oceanstore [29], Cleversafe's first commercial dispersed storage system, and Microsoft Azure's implementation of the LRC erasure code [9].

Cauchy Reed-Solomon codes also allow one to employ values of w that are not powers of two. For example, one may choose $w = 3$, and multiply a region of 3 KB by a constant in $GF(2^3)$ by partitioning the region into three packets of 1 KB each. Blömer *et al* recommend using the smallest possible value of w to achieve the best performance [2].

8.6.2 Leveraging Vector Instructions

When one implements $GF(2^{16})$ using the *SPLIT-TABLE* implementation with $g_a = 4$ and $g_b = 16$, each multiplication requires four table lookups, where each table holds sixteen 2-byte words. To leverage **mm_shuffle_epi8()**, we need to employ tables that hold sixteen bytes rather than sixteen words. A natural way to do this is to split each 2-byte table into two 1-byte tables. The first holds the high bytes of each product, and the second holds the low bytes. We name them T^i_{high} and T^i_{low} for $i \in \{0, 1, 2, 3\}$.

A straightforward way to employ these tables is to perform eight table lookups per 128-bit vector, as pictured in Figure 23. To match the SIMD architectures, in this picture, we draw the low bytes and sub-words of each 16-bit word on the right, and the high bytes and sub-words on the left. Only four of the eight words in the vector are shown, and each 16-bit word is shown partitioned into four 4-bit words, which are used as indices for **mm_shuffle_epi8()** table lookups. Each set of four-bit indices is used for two table lookups — one in the *low* table, and one in the *high* table.

If we employ an alternate mapping, we may improve the performance of this operation by roughly a factor of two. Consider the mapping where $x = 2$ and $y = 8$. Each 16-bit word is split into two bytes, and each byte is stored in a different vector. Thus, every set of 16 words is split over two 128-bit vectors. One vector stores the high bytes and one stores the low bytes. Now, we may use the same eight tables, T^i_{high} and T^i_{low}, as before, and perform 16 bytes worth of table lookups per **mm_shuffle_epi8()** operation, rather than eight as in Figure 23.

Figure 24 shows the process. In this figure, we perform 32 bytes of multiplication with eight **mm_shuffle_epi8()** operations, as opposed to 16 bytes in Figure 23. We call this the "alternate" mapping for *SPLIT-TABLE*, or "Altmap" for short.

If the standard mapping is required, for example because of backward compatability, it is faster to convert each 32-byte region to the alternate mapping, multiply them using the alternate mapping, and convert them back to the standard mapping. The exact SIMD instructions to perform this conversion are detailed in [23].

In $GF(2^{32})$, we may use *SPLIT-TABLE* with $g_a = 4$ and g_b = 32, and an alternate mapping with $x = 4$ and $y = 8$. Each 32-bit word is then split across four 16-byte vectors, and 32 lookup tables are required to multiply each 64-byte chunk of data. As with $GF(2^{16})$, it is faster to convert the standard mapping to the alternate mapping and back again if the standard mapping is required for an application.

This technique extrapolates to $GF(2^{64})$ as well, although the number of lookup tables blows up to 128.

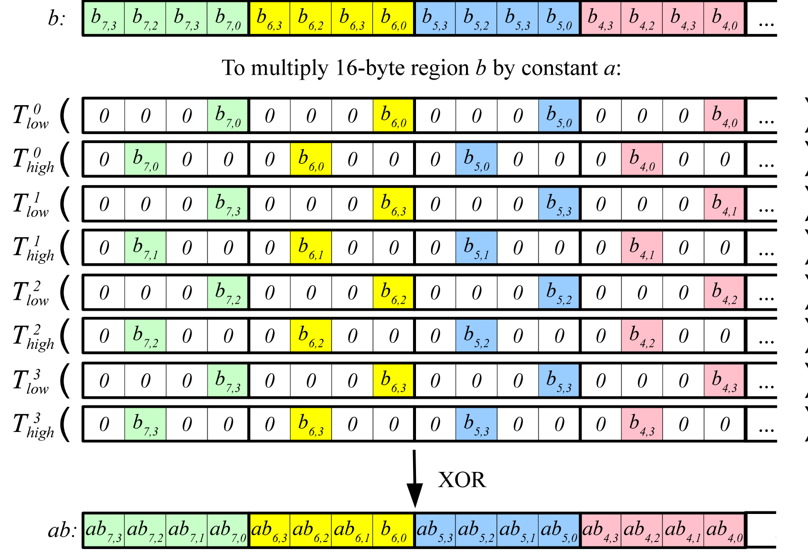


Figure 23: Using **mm_shuffle_epi8()** to multiply a 16-byte vector b by the constant a in $GF(2^{16})$. Each small box is a four-bit word, and each 128-bit vector holds words b_0 through b_7.

8.6.3 Leveraging recursive multiply_region() calls in *COMPOSITE*

As described in Section 6.8, the *COMPOSITE* implementation employs recursive calls to a smaller Galois Field. However, if the standard mapping is employed, one cannot implement **multiply_region()** by recursively calling **multiply_region()** on the smaller field, because the elements in the smaller field are interleaved. Instead, when implementing $GF((2^l)^k)$, one may use an alternate mapping where $x = k$ and $y = l$. In that way, when implementing **multiply_region()**, one may recursively call **multiply_region()** on the smaller field.

We illustrate with a concrete example. In this example, we call **multiply_region(**0xa4c5**,** b**,** p**,** 1024**,** add**)** in $GF((2^8)^2)$. An irreducible polynomial for this field is $x^2 + 3 + 1$. Thus, when we want to perform $p = ab$ in $GF((2^8)^2)$, we split each number into two elements in $GF(2^8)$, and calculate both parts of p with Equations 8 and 9 from Section 6.8:

$$p_1 = a_1b_0 + a_0b_1 + 3a_1b_1$$
$$p_0 = a_0b_0 + a_1b_1$$

If we employ the standard mapping to represent elements of $GF((2^8)^2)$, then we are forced to implement **multiply_region(**0xa4c5**,** b**,** p**,** 1024**,** add**)** with the equivalent of calling **multiply()** 512 times. However, if we employ the mapping where $x = 2$ and $y = 8$, then we may implement **multiply_region(**0xa4c5**,** b**,** p**,** 1024**,** add**)** with the following recursive calls to **multiply_region()** on the subregions of b and p in $GF(2^8)$:

```
multiply_region(0xc5,                 b,        p,        512,  add)
multiply_region(0xa4,                 b + 512,  p,        512,  1)
multiply_region(0xa4,                 b,        p + 512,  512,  add)
multiply_region(0xc5,                 b + 512,  p + 512,  512,  1)
multiply_region(multiply(3, 0xa4),    b + 512,  p + 512,  512,  1)
```

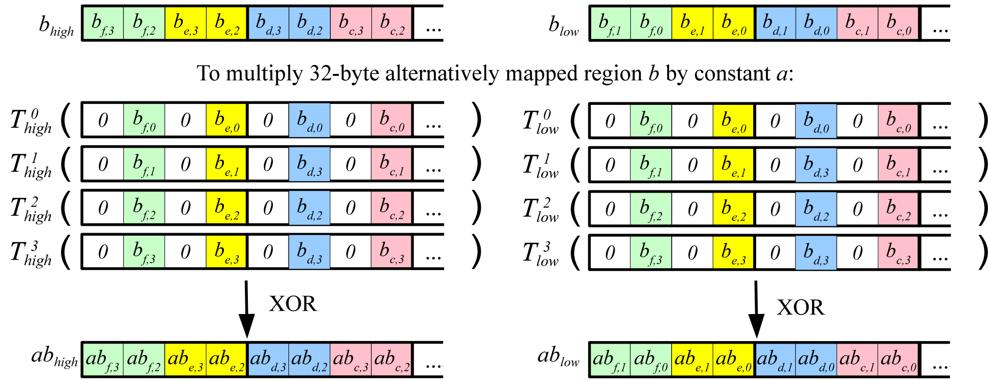


Figure 24: Using an alternate mapping to multiply two 16-byte vectors b by the constant a in $GF(2^{16})$. Each pair of 128-bit vectors holds words b_0 through b_f. Each word is split into two bytes, each of which is stored in a different vector.

Thus, instead of performing $512 * 5 = 2560$ separate multiplications and $512 * 2.5 = 1280$ XORs, we simply perform **multiply_region** five times on 512-byte regions. The alternate mapping therefore can provide significant performance improvements.

8.7 Region Multiplication Summary

To summarize, we have detailed the following additional performance improvements for implementing multiplication of a region by a constant.

Technique	Performance improvement
SHIFT	None
GROUP	Lazy Table Creation;
BYTWO$_p$	Anvin's Optimization
BYTWO$_b$	Anvin's Optimization; Alternate mapping (Cauchy Reed-Solomon)
TABLE	Small Table Lookup; Quad-Tables; Double-Tables; Lazy Table Creation
LOG-TABLE	Only one table lookup
SPLIT-TABLE	Small Table Lookup; Lazy Table Creation; Alternate Mapping
COMPOSITE	Alternate Mapping

9 Implementation

We have implemented an open-source library called "GF-Complete" that implements all of the techniques described in this paper. It is open source and available on Bitbucket at https://bitbucket.org/jimplank/gf-complete. All of the performance numbers in this paper come from this implementation.

10 Performance

In this section, we perform a rough evaluation of the performance of GF-Complete. The evaluation is "rough" because there are too many variations in terms of parameters, machines, configurations and usage scenarios to perform a complete evaluation. Moreover, Greenan *et al* have demonstrated that machine architectures and usage scenarios affect the impact of Galois Field implementation in ways that are very difficult to quantify [6]. Therefore, our intent in this section is to convey a sense of how the implementations perform with respect to basic microbenchmarks, with the understanding that in more complex systems, the performance of the library impacts the system in ways that are difficult to quantify.

All of our tests are performed on a standard commodity microprocessor – an Intel Core i7-3770 running at 3.40 GHz – running GNU/Linux in user mode, but in isolation. The machine has 16 GB of RAM and an 8MB Intel Smart Cache, consisting of 4 x 256 KB L2 caches, and an 8 MB L3 cache. All tests are run on one core, and all SIMD operations are supported.

10.1 Multiplication

To test multiplication, we populate two 64 KB regions of memory, A and B, with random numbers. We chose that size because all of the data fits into the L2 cache. For $w = 4$ and $w = 8$, this results in 64K random numbers per region, and for the larger w, it results in $\frac{64K(8)}{w}$. We multiply each number $a_i \in A$ with the corresponding $b_i \in B$, and then repeat the process 5,000 times.

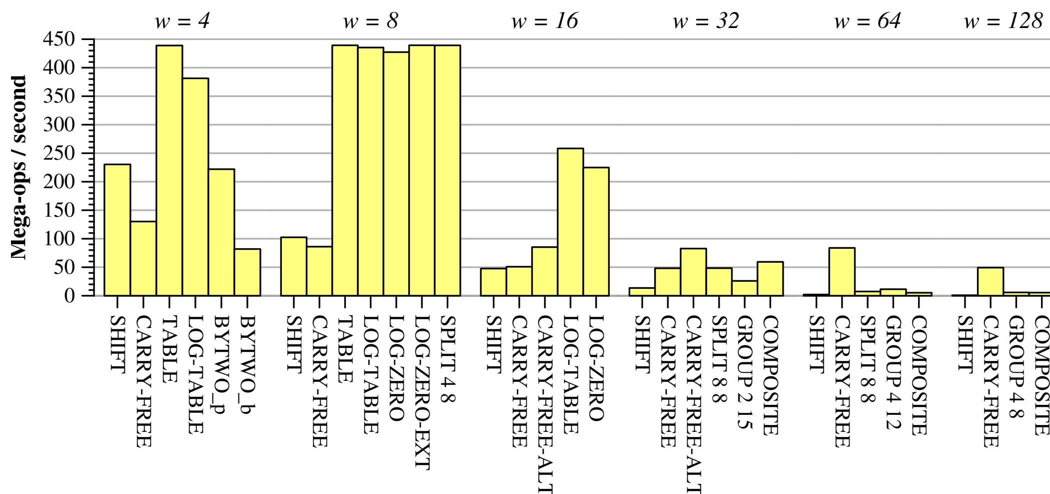


Figure 25: The performance of **multiply()** for a wide variety of implementation techniques.

We show the timing results in Figure 25. In the figure, we show results for all values of w, with the implementation techniques ordered according to their presentation in Section 6. We omit the *BYTWO* implementations for values of w greater than 4, because they always perform worse than *SHIFT*.

The best performing implementations are the *TABLE* implementations for $w \in \{4, 8\}$ and the *LOG-TABLE* implementations for $w \in \{8, 16\}$. For $w \geq 32$, the *CARRY-FREE* implementation significantly outperforms the

others; however it is important to select a primitive polynomial that allows the reduction to complete in two steps. For $w \in \{16, 32\}$, these are the alternate primitive polynomials from Figure 5, which are denoted *CARRY-FREE-ALT* in Figure 25.

For the *GROUP* implementations in $w \geq 32$, we enumerated values of g_m and g_r until we saw declining performance, and we plot the best of these, with the values of g_m and g_r in the axis labels. As noted in Section 6.7, since the multiplication table is created anew for each multiplication, and the reducing table is created once overall, the best combination has g_m be smaller than g_r.

For $w = 32$, the fastest non-SIMD implementation is *COMPOSITE*, which uses *LOG* as the implementation in the base field. The fastest non-SIMD implementation of a "standard" Galois Field for $w = 32$ is the *SPLIT* implementation where $g_a = g_b = 8$, which employs seven 256*256 tables. For $w \in \{64, 128\}$, the *GROUP* implementation is the best implementation without SIMD instructions.

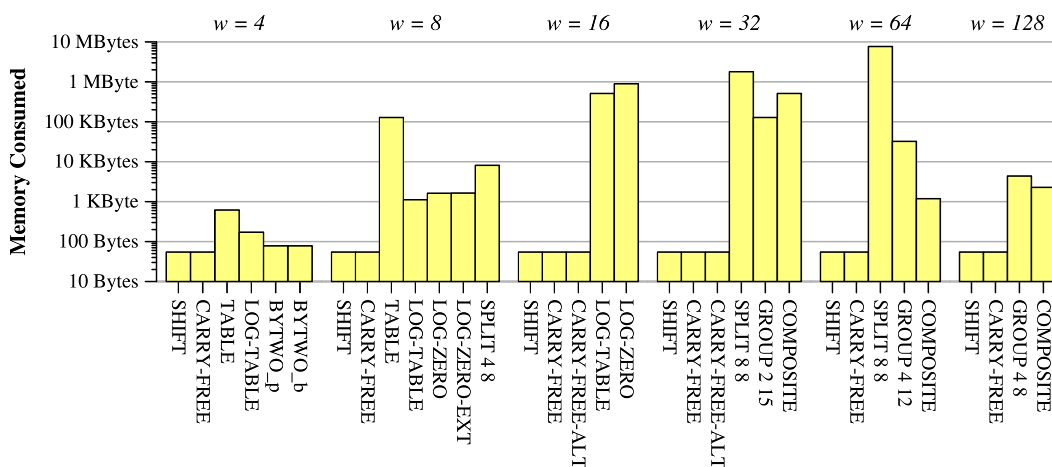


Figure 26: The memory consumption of each implementation technique.

In Figure 26, we plot the memory consumption of each implementation technique. The Y-axis is a log scale, meaning each horizontal bar is a factor of ten. Coupled with Figure 25, we can conclude that memory has a significant impact on performance. For example, in terms of instruction count, *TABLE* is better than *SPLIT-TABLE*, which is better than *LOG-TABLE*. However, in $GF(2^8)$, all three implementations perform similarly, because *TABLE* consumes much more memory than *SPLIT-TABLE*, which in turn consumes more memory than *LOG-TABLE*. In a similar vein, although *LOG-ZERO* requires fewer instructions than *LOG*, its performance is worse when $w = 16$ because of its increased memory requirements.

10.2 Division

When *TABLE* and *LOG-TABLE* are employed, the speed of division is identical to multiplication. The other implementations require either Euclid's method or inverting a bitmatrix. In GF-Complete, the bitmatrix inversion is significantly slower than Euclid's method, so we do not include its timings. In Figure 27, we show the fastest division times for each value of w when Euclid's algorithm is employed. The testing methodology is the exact same as for multiplication, except we divide a_i by b_i, and we never allow b_i to be zero.

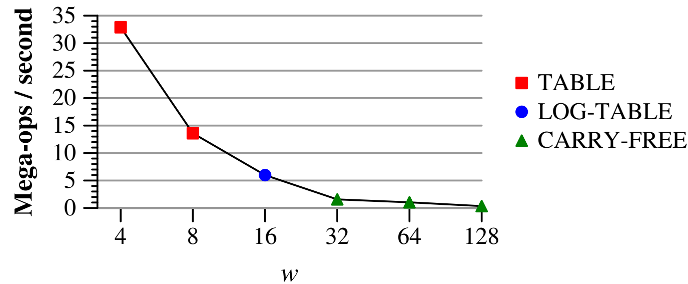


Figure 27: The fastest implementations of Euclid's algorithm for division.

10.3 Multiplying a Region by a Constant

To test the performance of **multiply_region()**, we repeatedly fill regions of memory with random numbers, and then time the calls to **multiply_region()**. We do this until a total 1 GB of regions have been multiplied. We test region sizes that are even powers of two, from 1KB to 1GB. Each data point is the average of over ten runs.

The performance of these operations is significantly impacted by the size of the regions as they relate to the various caches. To demonstrate, Figure 28 shows the performance of three basic operations as the region sizes are varied. The first is **memcpy()**; the second is XOR and the third is multiplying a region by two using Anvin's optimization as described in Section 8.5. The latter two operations are implemented using the Intel SIMD instructions, so they proceed 128 bits at a time. For multiplication by two, we plot the performance in $GF(2^4)$; however, the code is identical in any Galois Field, except for the primitive polynomials and the masks.

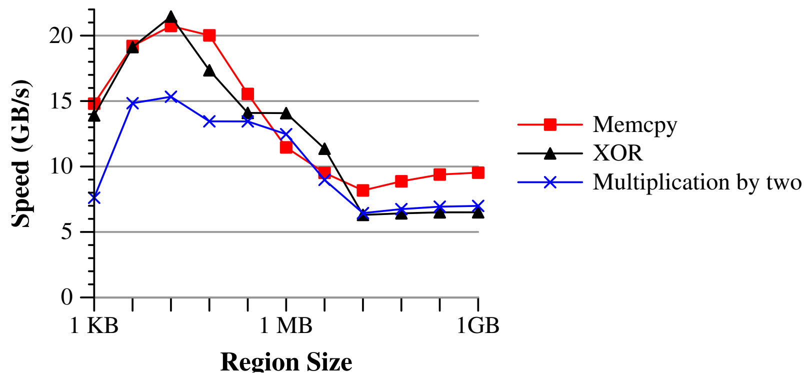


Figure 28: The performance of **memcpy()**, XOR, and multiplying a region by two in a Galois Field using SIMD instructions.

The three curves have similar shapes. As the region sizes increase from 1K to 16K, the performance improves as startup costs are amortized. As the region sizes increase further, the performance drops as the L2 (**memcpy()** and XOR), and then the L3 caches (all three curves) are saturated. At that point, the performance is more than a factor of two slower than the peak performance.

In the graphs that follow, we plot peak performance for the region size that performs the best. When the performance

exceeds roughly 6 GB/s, the performance is limited by the L3 cache. When the performance is less than that, then it is limited by the instruction.

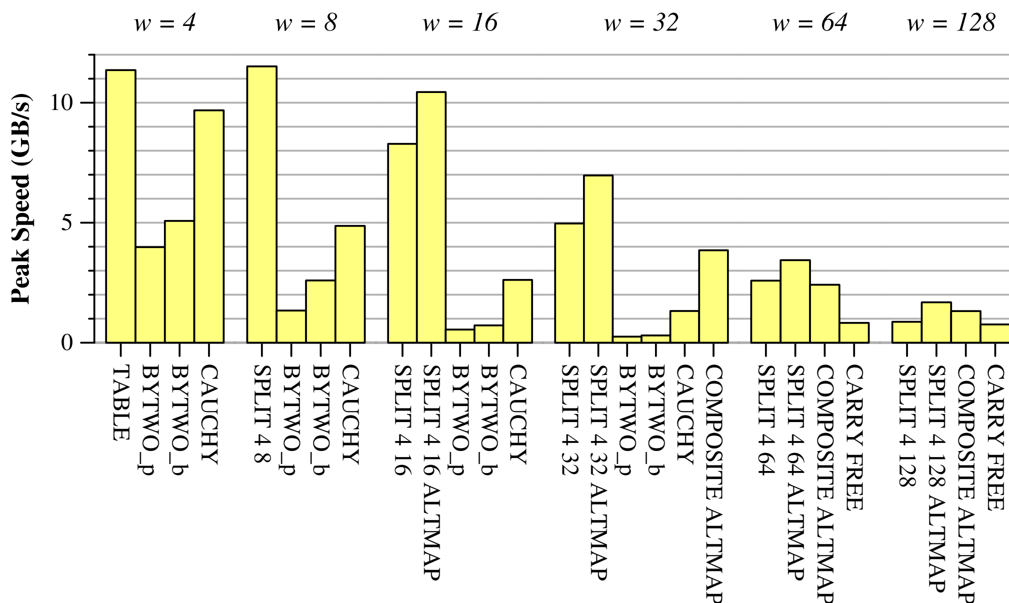


Figure 29: The peak performance **multiply_region()** with SIMD instructions.

In figure 29, we show the peak performance of **multiply_region()**, when the implementations leverage SIMD instructions. For every value of w, the best performance is achieved by leveraging **mm_shuffle_epi8()** with either *TABLE* ($w = 4$) or *SPLIT-TABLE* (the other w). For $w > 8$, the alternate mapping of words to memory in *SPLIT-TABLE* outperforms the standard mapping significantly, as the alternate mapping allows us to better leverage **mm_shuffle_epi8()**. In particular, when $w \leq 32$, the alternate mappings are cache limited.

When $w = 4$, the *CAUCHY* implementation, which relies solely upon XOR, is competitive with the *TABLE* implementation. This is the implementation technique used by Microsoft in their Azure storage system [9]. The implementation does not leverage intermediate results, nor does it pay explicit attention to cache optimization [16, 25]. Were it to do so, its performance may exceed *TABLE*. As w grows, the performance of *CAUCHY* declines because it has to perform relatively more XOR operations. For the values of w greater than 4, *CAUCHY* is a less attractive alternative.

Even though the *BYTWO* implementations operate on 128-bit words, their performance is worse than *TABLE* and *CAUCHY*. The *COMPOSITE* tests in Figure 29 use the best performing region operations in their respective base fields. Although they perform well, they never outperform the implementations based on **mm_shuffle_epi8()**. They also require an alternate mapping of words to memory so that they may leverage recursive **multiply_region()** calls. We only show *CARRY-FREE* results for $w \in \{64, 128\}$. In neither case is the implementation competitive with the other implementations.

In figure 30, we show the peak performance of **multiply_region()** without SIMD instructions. The speeds in this figure are much slower than in figure 29. In all cases except $w = 4$, the table-based implementations perform the best, and with the exception of $w = 32$ and $w = 128$, tables indexed by 16-bit indices performed the best. Because these

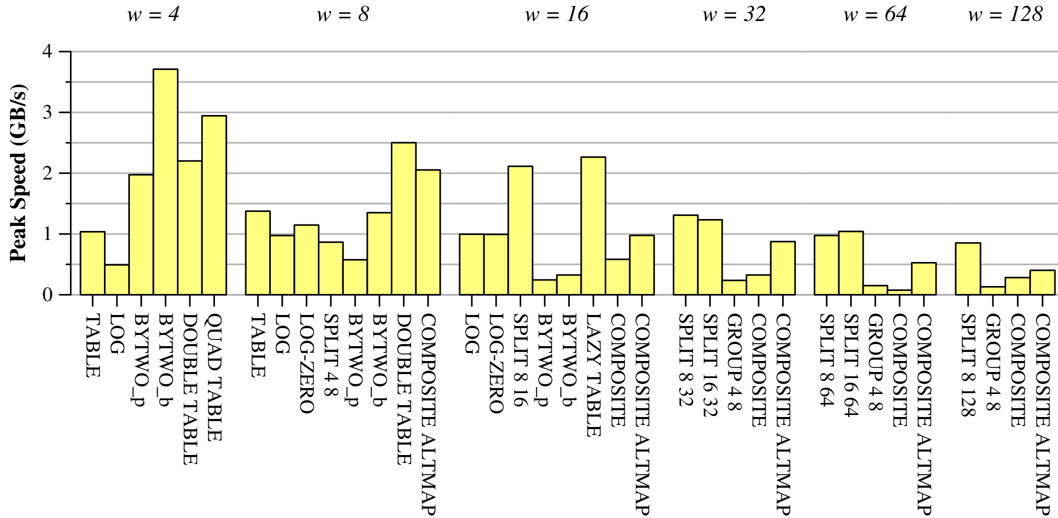


Figure 30: The peak performance **multiply_region()** without SIMD instructions.

tables are large, they are created lazily, and thus perform best on large regions that can amortize the table creation. With $w = 32$, 8-bit tables outperform the 16-bit tables slightly. With $w = 128$, we did not implement 16-bit tables, because they would require 8MB of tables to be initialized for each **multiply_region()** call.

When $w = 4$, the *BYTWO*$_b$ implementation performs the best, as it works on 64 bits at a time, and requires very few iterations. Unlike *BYTWO*$_p$, it can terminate iterating early, depending on the value being multiplied; hence, it performs better than *BYTWO*$_p$ in all instances. As in Figure 29, the *ALTMAP* versions of *COMPOSITE* use the best implementations of **multiply_region()** in the base fields. The implementation for $w = 8$ is interesting because unlike the 16-bit table, which requires 128K of memory, the *COMPOSITE* implementation uses *BYTWO*$_b$ in the base field, and therefore has no extra memory consumption.

We stress again that these numbers are machine specific, and only give a rough picture of how these implementations perform in general.

11 Conclusion

In this paper, we have detailed a wide variety of techniques to implement Galois Field arithmetic for erasure coding applications. Many of them leverage SIMD instructions which are found in today's microprocessors, and these perform at speeds that are limited by how fast the machines' caches may be populated. This paper is accompanied by an open source library called GF-Complete.

12 Acknowledgement

The authors would like to thank Adam Disney, Allen McBride, Ben Arnold and John Burnum for co-authoring the implementation of GF-Complete, and Will Houston for doing the early work on GF-Complete.

References

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

[2] 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.

[3] 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 (SOSP '11)*, October 2011.

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, Boston, third edition, 2009.

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

[6] 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.

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

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

[9] 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.

[10] 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.

[11] D. Kenchammana-Hosekote, D. He, and J. L. Hafner. REO: A generic RAID engine and optimizer. In *FAST-2007: 5th Usenix Conference on File and Storage Technologies*, pages 261–276, San Jose, February 2007.

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

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

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

[15] 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.

[16] J. Luo, M. Shrestha, L. Xu, and J. S. Plank. Efficient encoding schedules for XOR-based erasure codes. *IEEE Transactions on Computing*, May 2013.

[17] J. Luo, L. Xu, and J. S. Plank. An efficient XOR-Scheduling algorithm for erasure codes encoding. In *DSN-2009: The International Conference on Dependable Systems and Networks*, Lisbon, Portugal, June 2009. IEEE.

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

[19] T. K. Moon. *Error Correction Coding: Mathematical Methods and Algorithms*. John Wiley & Sons, New York, 2005.

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

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

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

[24] 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.

[25] J. S. Plank, C. D. Schuman, and B. D. Robison. Heuristics for optimizing matrix-based erasure codes for fault-tolerant storage systems. In *DSN-2012: The International Conference on Dependable Systems and Networks*, Boston, MA, June 2012. IEEE.

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

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

[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] J. H. van Lint. *Introduction to Coding Theory*. Springer-Verlag, New York, 1982.