12-2015

Grouping Digits As a Method For Increasing Computation Speed Of Galois Field Arithmetic For Erasure Coding Applications

John Andrew Burnum
jburnum@vols.utk.edu

Grouping digits as a method for increasing computation speed of Galois Field arithmetic for erasure coding applications

John Burnum, Faculty Advisor: James Plank

***Abstract*—The performance of multiplications in Galois Field arithmetic using the GROUP method is measured, described, and analyzed using the GF_Complete open-source library. Galois Field arithmetic is vital for many erasure coding methods, including Reed-Solomon coding, and the GROUP method is a simple and modular technique for increasing computation speed.**

I. Introduction

Information storage is not a modern problem. The invention of writing and every innovation on it have, sometimes slowly and other times quickly, increased society's ability to store information to be retrieved later. With the advent of modern computers, the ability to store, copy, and modify large amounts of data very rapidly has become commonplace. However, one cost of electronic data storage is its fragility. More primitive forms of information storage can be destroyed, but are less prone to spontaneous failure. Appropriately, as the use of electronic data storage has become ubiquitous, techniques for preventing loss of data have become more advanced. The most prevalent family of techniques for protecting data is known as ”erasure coding.” Erasure coding ensures that if a portion of the data is ”erased” for whatever reason, the whole body of data can still be reproduced. An important note is that erasure coding assumes all failures are ”erasure” failures, where it is readily apparent that the data is missing and from where the data is missing. An example of failures which don't follow this assumption is if the existing data is modified to be incorrect, but is still readable.

The simplest version of erasure coding is duplication of data. If you have two copies of the data and you lose one, then you still have another copy which represents the whole accurately. However, duplication is very expensive. For example, assume a certain body of data fills up four disks. To duplicate it, another four disks must be used. If any one of these eight disks fails, then the full data is recoverable. However, if two disks fail, it is possible for corresponding disks in each set of four to fail, which constitutes to an irrecoverable loss of data. So full duplication only fully protects against a single disk loss. A different way to protect against a single disk loss is to bitwise XOR the data in the four disks and to store the result in a fifth disk [**?**]. Then, if any one of the disks fails, the data on it can be reconstructed simply by performing the bitwise XOR of the remaining disks. Compared to duplication, this method provides the same protection against any single disk loss with much less additional storage, using only one extra disk instead of four extra disks. However, it does require some amount of computation to set up the data for storage and to recover the data after a failure. The set up computation is known as ”encoding” and the recovery computation is known as ”decoding.” For more complex schemes which protect more data, it is necessary to do far more computation to encode and decode.

Erasure coding follows a few general conventions. First we assume that we have n storage disks. We have k disks' worth of data to be stored. We then have the other $n - k = m$ disks of space which are used for encoding data. In what are called systematic codes, the original data is stored on k of the disks, and the other m disks are encoding data [**?**]. Appropriately these are referred to as data disks and coding disks, respectively. However, there are also non-systematic codes. In non-systematic codes, the user provides k disks' worth of data, and the code stores n disks' worth of data, just as with systematic codes. However, non-systematic codes don't necessarily store a copy of the original user data. The original user data is obviously recoverable, but it is all encoded and must be decoded before it can be read. In either case, the user chooses k disks' worth of data, and the code stores n disks' worth of data. The content of the n disks is calculated based on the user's original data.

Another convention is that erasure coding labels a certain minimum size of data as a ”word.” This label is convenient because erasure coding uses special forms of arithmetic called finite field arithmetic. The advantage of finite field arithmetic is that calculated results are guaranteed to be convenient to store in the same amount of storage while still maintaining reversibility of operations. In conventional arithmetic, operations are reversible: You can multiply by any integer and then divide by the same integer and end up with the same number. However, if you multiply two integers that each fit in four bytes, you are not guaranteed that their product will fit in four bytes, and additional logic to deal with the overflow is expensive. Finite field arithmetic avoids this problem by allowing values to ”wrap around” the maximum value in a certain way that ensures that each number still has an inverse.

Mathematically, a field is a set of elements which has two defined operations. These operations are analogous to addition and multiplication in conventional arithmetic. In fact, conventional addition and multiplication on the rational numbers forms a field. However, there are a few more requirements for the elements and operations to form a field. First, the operations must both be closed on the set of elements. That is, the sum or product of any two of the elements in the set must be another element in the set. This seems simple enough for infinite fields such as the integers, but for finite fields it requires very unconventional operations. In finite fields, repeated sums must not become continuously larger numbers, because that requires an infinite field for the sums. Second, a multiplicative and an additive identity must exist. That is, there is an element a and an element b such that

$$a * x = x \quad \forall x$$
$$b + x = x \quad \forall x$$

The additive identity is also called the zero element. Thirdly, each element of the set which is not the additive identity must have a multiplicative and an additive inverse. That is, for each element in the set, there exists an element in the set such that their sum is zero, and there exists an element in the set such that their product is one. The existence of inverses in the field is required for operations to be invertible. If operations are invertible, an equation can be solved by simply applying the inverse to each side of the equation.

What makes a finite field different from more familar fields is simply that it has a finite number of elements. This allows any element in the set to be represented by the same data type. For example, for a four byte data type, we can represent any element in a set which has up to $2^{(8*4)}$ elements. This guarantees that any operation or inverted operation produces a value which can still be stored in a four byte computer data type. The most commonly used type of finite field in erasure coding is called a Galois Field. Galois Fields are defined by a whole number w and what is called a ”primitive polynomial.” Galois Fields have 2^w elements, which makes them convenient for binary implementations. Each element can be stored in w bits; so Galois Field-based erasure codes have words of w bits in length. The analogue for addition in Galois Fields is bitwise XOR, which is very fast on conventional computer hardware. The analogue for multiplication is more complicated. In Galois Fields with $w = 1$, it is the equivalent of bitwise AND. For larger values of w, the multiplication analogue requires two steps: first, a multiplication step, and then a simplification step. The multiplication step may be computed similarly to conventional long multiplication except with bitwise XOR instead of addition. To be more precise, to multiply two numbers X and Y with binary digits $x_1, x_2 \dots x_n$ and $y_1, y_2, \dots y_n$,

$$X * Y = \bigoplus_{i=1}^{n} (X \wedge y_i) << i$$

where $\oplus$ refers to bitwise XOR, $\wedge$ refers to bitwise AND, and $<<$ refers to left bitshift. This value will be referred to as the ”partial product.” Because the partial product can be a value outside the field, it must be reduced down to a value inside the field. The primitive polynomial is the tool for reducing the partial product. This polynomial can be represented by a value which is outside the field. In binary representation, it has $w + 1$ digits, and the highest value digit is a 1. The partial product is reduced by computing the bitwise XOR of the partial product and the bit-shifted primitive polynomial. Specifically, the primitive polynomial is bit-shifted to align the highest value 1 with the highest value 1 in the partial product. This process is repeated until the partial product is reduced down to a value within the field, at which point it is the final product. A useful observation is that the entire reduction step can be described as performing the multiplication step between the excess digits and the primitive polynomial, and then performing the XOR of this result with the partial product. With appropriate choice of primitive polynomial, this allows for reversibility of operations and so the Galois Field is truly a field. Here’s an example of the arithmetic in $w = 4$:

				1	1	0	0
			x	0	1	1	0
				0	0	0	0
			1	1	0	0	
		1	1	0	0		
$\oplus$	0	0	0	0			
	0	1	0	1	0	0	0

Above is an example of the computation of the partial product of 1100 and 0110 in $w = 4$. The partial product is then outside of the Galois Field and needs to be reduced back to within the field using the primitive polynomial. The primitive polynomial for $w = 4$ that we have chosen is 10011. Below is the same example carried through the reduction steps to reach the final product.

	0	1	0	1	0	0	0
$\oplus$		1	0	0	1	1	
				1	1	1	0

Once a value of w is selected, the organization of data on the disks becomes better defined. Each disk is divided into w-bit ”words.” Each of these words is entered into the encoding computations independently of its neighbors, so the encoding is done in parallel. For example, the simple case of having one encoding disk which is equal to the bitwise XOR of the other disks can be described as a Galois Field in $w = 1$, so each word is only 1 bit. In this example, the generalization to more encoding disks is not trivial at all. In fact, an entirely new scheme is required. A next step is to define a second coding disk. Its contents are set to

$$\bigoplus_{i=1}^{k} 2^i * d_i$$

where d_i is the data stored on the k data disks. This setup is then able to tolerate the failure of any two disks. Most erasure codes are defined in a similar fashion, such that each coding disk is a linear combination of the data disks in Galois Field arithmetic. These codes are then completely determined by the Galois Field they use and by the coefficients of the linear combinations. In fact, erasure codes can be represented conveniently by matrices with these coefficients as the elements. For example, the code being discussed with k=4 would appear as:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 2 & 4 & 8 \end{bmatrix} \times \begin{bmatrix} d_1 \\ d_2 \\ d_3 \\ d_4 \end{bmatrix} = \begin{bmatrix} c_1 & c_2 & c_3 & c_4 & c_5 & c_6 \end{bmatrix}$$

Where each d_i is one of the k data disks, and each c_i is one of the n total disks. As demonstrated above, the encoding process is then simply taking the product of the matrix with the k disks of data to produce the n disks of encoding. Decoding involves simply modifying the matrix by removing rows corresponding to erased sections of encoding and then inverting the encoding process. To be precise, the matrix is then inverted and the product of the inverted matrix with the surviving sections of encoding is computed. This product is the original data, and decoding is complete.

The practical differences between different erasure codes are the costs and the levels of protection. To describe levels of protection, we first define maximum distance separable (MDS) codes. MDS codes are erasure codes that protect from any combination of failures in units of words as long as the original amount of data survives. For example, if we have one coding disk which is set as the XOR of the other disks, then we are protected from the failure of any one disk. Therefore, this code is an MDS code [?]. Interestingly, though, this code is arguably applied to each bit on a disk in parallel, and so it is each corresponding set of bits which is stored in an MDS code, not the disks as a whole. This leads to a different set of failures which are protected against. Similarly, the setup with two linearly independent coding disks can tolerate the failure of any two disks; it is also an MDS code. Both of these schemes can protect against any selection of individual disk failures, up to their maximum tolerances. An example of a non-MDS code is simple replication, so that each portion of the data has two copies. Even considering only disk failures, it is possible for corresponding sections of the data in each copy to fail and become erased. In this way, replication is not an MDS code. Codes which are MDS correspond to encoding matrices in which every combination of rows in which the total number of rows is equal to the number of columns in the matrix is invertible.

To discuss storage costs for erasure codes, it should be noted that the definition of MDS is closely tied to storage cost. MDS codes by definition protect against as many failures as there is storage space for coding. However, computation costs vary significantly between erasure codes, even MDS codes. Non-systematic codes even require some computation just to read data. However, even systematic codes have their own bottlenecks [?]. Whenever the encoded data is modified, the encoding has to be modified. If the encoding storage is isolated from the data storage, then the encoding storage will have to be accessed and written to far more often than any of the data storage. Certain forms of MDS codes, known as Reed-Solomon codes, are easy to derive for any selection of parameters [?], [?], [?]. This means that the algorithms for any scale of data protection system are readily available. However, Reed-Solomon codes, which are based on Galois Field arithmetic, require higher values of w for larger scale systems. At the same time, Galois Field arithmetic becomes more complex for higher values of w [?]. Another option is erasure codes which allow for extremely fast encoding and decoding, but aren't MDS. Because of all the different factors and relationships between costs and benefits, techniques to improve encoding and decoding rates by reducing computation complexity are important to practically implement erasure coding data protection.

II. GROUP Method

There are several techniques for improving the performance of Galois Field arithmetic [?], [?], [?]. There are hardware implementations which allow doing multiple operations in parallel. There are mathematical techniques in finite field arithmetic which can create shortcuts in calculations. However, the most generally applicable algorithms involve only doing computations once and then building a table in memory where appropriate values can be looked up when needed. One version of this technique, known as the GROUP method, simply groups two or more digits together into a single unit so that computations can proceed two or more times faster. This technique relies on the fact that certain computations will have to be done many times in the course of completing a single encoding or decoding. If the computations are not repetitive, then it becomes prohibitively expensive to commit enough memory to be able to significantly speed up computations. Conveniently, multiple techniques of erasure coding lead to repetitive computations. Both multiplying large regions of data by the same coefficient and continuously using the same primitive polynomial to reduce partial products are examples of repetitive computations. The method can use one table to repeatedly multiply by the same coefficient and another to repeatedly reduce in the same Galois Field, so it is simple to leverage the patterns of conventional erasure coding.

Because Galois Field multiplication is conventionally calculated using a method which is very similar to long multiplication in standard arithmetic, the GROUP method is easy to describe analogously. In the context of doing standard arithmetic by hand, multiplying an 8-digit number by a 3-digit number is inconvenient. To do long multiplication, one multiplies a single digit by a single digit at a time and then sums the partial products. This leads to a total of $3 * 8 = 24$ multiplications of single-digit pairs. Then, similarly, there are 23 addition operations, though many of these are handled as the multiplication occurs through carrying digits. As these partial products calculated and the sums arranged, some of the products are shifted over a certain number of digits before they are summed. This is to account for where the digits appear in the factors, to make sure that the numbers deliver their true magnitude into the product. So the GROUP method involves grouping these digits. In our analogy, we could take the 3-digit number and calculate the product of it and each of the ten single digits. We will write all ten of these products in a table. Once we have this table, we only have to do 8 table look-ups and 7 additions to calculate our total product. However, in this specific example, doing the 30 multiplications and 20 additions to build the table obviously isn't worth the gains from using the table. On the other hand, if the 8-digit number were instead a 100-digit or 10,000-digit number, then the gains due to building the table first would be very large. Similarly, the GROUP method calculates the products of one factor and each possible combination of digits. These are stored in a table, and then table look-ups are used instead of multiplications. Also similarly, the method becomes more and more efficient as the number of operations the table can be used for increases.

Here is the algorithm in more detail. The algorithm has two parameters: g_m and g_r. g_m is the number of digits grouped together during the multiplication step, and g_r is the number of digits grouped together during the reduction step. Both the construction of the multiplication table and the reduction table are straightforward; the description will begin with the multiplication table. Because the encoding and decoding processes often involve multiplying regions of data by the same coefficient, the table is built so that the indices into the table are every possible combination of g_m binary digits. The value stored in the table is then calculated as the full product, including any needed reduction steps, of those g_m digits and the coefficient. Similarly, the reduction table is constructed with the indices into the table being every possible combination of g_r binary digits, and the values in the table are the full products between the index and the primitive polynomial. An example of a reduction table in $w = 4$ follows:

0	0	0	0	0	0	0	0
0	1	0	1	0	0	1	1
1	0	1	0	0	1	1	0
1	1	1	1	0	1	0	1

To use these tables to assist in multiplication, simply divide the data into groups of g_m digits. For each group, look up the product in the table, then bitshift the product the appropriate amount, then XOR the products together. Specifically the appropriate amount of bitshifting is as follows: For the ith product, counting from lowest-order binary digits as the 0th group to highest-order, bitshift it $i * g_m$ bits. For the reduction step, divide the excess higher-order bits into groups of g_r digits each, and look up the corresponding products in the table. Then bitshift the products appropriately (identical to the method for the multiplication step) and XOR them with the partial product.

In summary, for every multiplication by that coefficient, the number of bitshifts is reduced significantly, the number of bitwise XORs is reduced by approximately a factor of g_m, and the number of bitwise ANDs is reduced to zero. All of these operations are replaced by w/g_m table look-ups [?]. These tables are only useful, of course, as long as the coefficient doesn't change. On the other hand, the reduction table is far longer-lived, because it is based on the primitive polynomial instead of the coefficients. The reduction table provides similar levels of gains in computation except that the number of excess digits needed to cancel with the primitive polynomial varies significantly, so on average the gains are slightly more than half as much as for the multiplication table. These tables have both a computation and memory cost, however. For each entry in the table the computational cost is one full multiplication in Galois Field arithmetic. Each entry in the table also costs w bits of memory. Then, the number of entries in the corresponding table doubles when g_m or g_r is increased by one.

There is also an additional optimization for when $g_m = g_r$. In this situation, instead of doing the entire multiplication step first, followed by the entire reduction step, the steps can be interleaved. The first g_m digits are looked up in the table as usual; then no reduction step is necessary. Then the second set of g_m digits are looked up in the table, bitshifted appropriately, and XOR'd with the first set. Then a reduction step is immediately done with the g_r excess digits. It is guaranteed that only a single reduction step is needed for each multiplication step because there are no carry digits, so the only excess digits are the ones from the g_m bitshifts. Because $g_m = g_r$, the excess digits are perfectly handled by a single g_r reduction.

III. Experimental Procedure

The speed of the GROUP algorithm was examined in the Galois Field where $w = 32$. This implies that the elements in the Galois Field can each be stored in 32 bits, or 4 bytes. The experiment focused on comparing the effects of g_m, which determines the size of the multiplication table, g_r, which determines the size of the reduction table, and region size on the speed of computation. Region size is important because a table is retained for an entire region, so larger region sizes will leverage the tables more. We performed the experiments by implementing the GROUP method in the C programming language, and then timing it on randomly created regions of data with randomly chosen coefficients. This means that a region of randomized data is built, and a random coefficient is also generated. Then the multiplication and reduction tables are built. Then the entire region is divided into 32-bit chunks, each of which is multiplied by the coefficient and then overwritten by the product. This tool produces a new multiplication table and a new reduction table for each region. Because the reduction table is rebuilt every region, the same rate at which the multiplication table is rebuilt, the main leveragable difference between g_m and g_r is not well-explored by this experiment. Instead, this allows direct comparison of the efficiencies of g_r and g_m within a specific product. Both g_m and g_r, the size of the tables, were varied from 2 to 16, inclusive. Also, the size of the regions being multiplied was varied from 2^{10} bytes to 2^{30} bytes in steps of powers of 2. The variation of these parameters allows an examination of the differences in effectiveness between multiplication table size and reduction table size at different region sizes. For lower size regions, the timing test was repeated enough times for the total running time to be significant in order to reduce the effect of random variance on the data.

The experiment was performed on a 3.4 GHz Intel i7-4770 processor with 32 KB of L1d cache, 32 KB of L1i cache, 256 KB of L2 cache, and 8192 KB of L3 cache.

Figure 1 compares the advantages of investing in the size of the multiplication table and the advantages of investing in

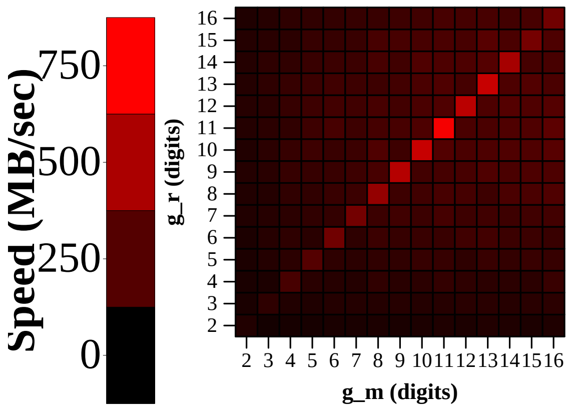

Fig. 1: Computation speed with varying table sizes in a region size of 1 MB

the size of the reduction table. Most notable is the advantage lent by the optimization when $g_m = g_r$. This condition, which is clearly visible along the diagonal, leads to much higher computation speeds than otherwise. There is also an apparent maximum at $g_m = g_r = 11$. Below this value, there is additional speed which can be acquired by building a bigger table, but above this value the speed quickly drops off as more time is spent building the table than on the actual computations within the region. This maximum shifts around as the region size changes, but in general the graphs at each region size look very similar to this one. Lower region sizes have peaks in computation speed at lower numbers of digits. For example, in a region size of 1 KB, the optimal number of digits was found to be 9 for this hardware. For every region size larger than 16 KB, 11 was the optimal number of digits. The reason that the optimal table size doesn't continue to increase is probably related to cache effects; if the size of the table ever becomes bigger than the cache size, then instantly the algorithm becomes slower and less efficient to perform the same calculations. This explanation makes sense for 11 to be the optimal number of digits. 11 digits in $w = 32$ means that the data in the table requires $4 * 2^1 1 = 8192$ bytes to store. There are two tables, one for multiplication and one for reduction, so together they require 16 kilobytes. To store enough for 12 digits in each table would require 32 kilobytes by themselves, so any other data (including the data to be multiplied) would defeat the advantages of the cache.

Figure 2 demonstrates the gains of investing in the size of the multiplication table as the region size varies. It was produced with g_r set to 2, which causes the noticeable speed-up along the bottom row, when g_m equals 2 as well.

Similarly, Figure 3 demonstrates the gains of investing in the size of the reduction table as the region size varies. Once again, it was produced with g_m set to 2, which causes the noticeable speed-up when g_r equals 2 as well.

The rest of the collected data continues these trends, so only a few of the possible charts are presented here.

IV. Conclusion

The GROUP algorithm is an effective method for speeding up Galois Field arithmetic. It is simple to understand and

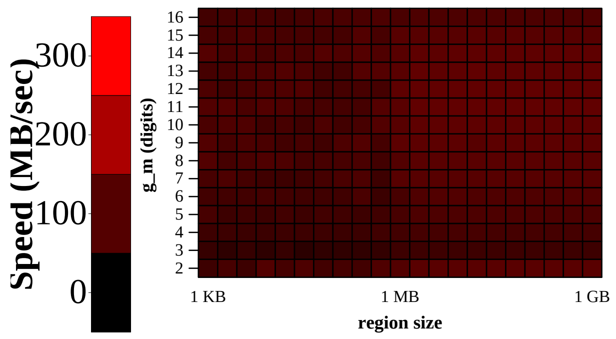

Fig. 2: Computation speed with varying multiplication table size and region size with $g_r = 2$

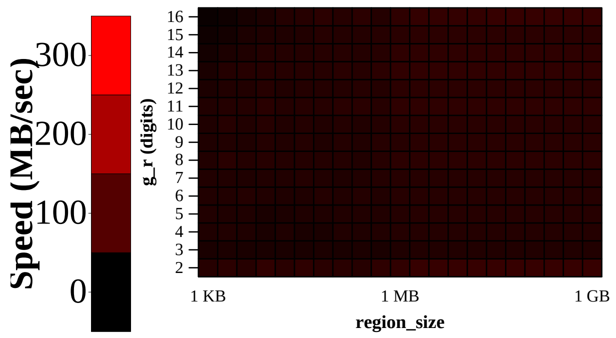

Fig. 3: Computation speed with varying reduction table size and region size with $g_m = 2$

implement, and has multiple parameters which can be tuned to suit the specific application. The fact that the method is based on storing shortcuts in memory makes the method dependent on how much memory is available. Even though main memory is not a practical restriction on the size of the table, the size of the cache provides an upper limit to effective gains from the size of the table. However, no matter the hardware used to implement the erasure coding system, there are still gains to be found using grouping of digits. The optimization which uses the same number of digits for the multiplication and reduction tables provides significant speed increases for implementations which are unable to leverage constructing a larger reduction table.

References

[1] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson, "RAID: High-performance, reliable secondary storage," *ACM Computing Surveys*, vol. 26, no. 2, pp. 145–185, June 1994.

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

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

[4] L. Rizzo, "Effective erasure codes for reliable computer communication protocols," *ACM SIGCOMM Computer Communication Review*, vol. 27, no. 2, pp. 24–36, 1997. [Online]. Available: http://doi.acm.org/10.1145/263876.263881

[5] 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.

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

[7] 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*, vol. 8, no. 2, February 2012.

[8] J. S. Plank, K. M. Greenan, and E. L. Miller, "A complete treatment of software implementations of finite field arithmetic for erasure coding applications," University of Tennessee, Tech. Rep. UT-CS-13-717, October 2013.