An Efficient XOR-Scheduling Algorithm for Erasure Codes Encoding

aaaaaaaaa

Jianqiang Luo, Lihao Xu
Wayne State University
jianqiang@wayne.edu, lihao@cs.wayne.edu

James S
University of
plank@cs

Abstract

In large storage systems, it is crucial to protect data from loss due to failures. Erasure codes lay the foundation of this protection, enabling systems to reconstruct lost data when components fail. Erasure codes can however impose significant performance overhead in two core operations: encoding*, where coding information is calculated from newly written data, and* decoding*, where data is reconstructed after failures.*

This paper focuses on improving the performance of encoding, the more frequent operation. It does so by scheduling the operations of XOR-based erasure codes to optimize their use of cache memory. We call the technique XOR-scheduling and demonstrate how it applies to a wide variety of existing erasure codes. We conduct a performance evaluation of scheduling these codes on a variety of processors and show that XOR-scheduling significantly improves upon the traditional approach. Hence, we believe that XOR-scheduling has great potential to have wide impact in practical storage systems.

1 Introduction

As the amount of data increases exponentially in large data storage systems, it is crucial to protect data from loss when storage devices fail to work. Recently, both academic and industrial storage systems have addressed this issue by relying on erasure codes to tolerate component failures. Examples include projects such as OceanStore [16], RAIF [15], and RAIN [6], and companies like Network Appliance [19], HP [27], IBM [9], Cleversafe [14] and Allmydata [1], employing erasure codes such as RDP [7], the B-Code [30] and Reed-Solomon Codes [5, 25].

In an erasure coded system, a total of $n = k + m$ disks are employed, of which k hold data and m hold coding information. The act of *encoding* calculates the coding information from the data, and *decoding* reconstructs the data from surviving disks following one or more failures. Storage systems typically employ *Maximum Distance Separable (MDS)* codes [4], which ensure that the data can always be reconstructed as long as there are at least k disks that survive the failures.

Encoding is an operation performed constantly as new data is written to the system. Therefore, its performance overhead is crucial to overall system performance. There are two classes of MDS codes – *Reed-Solomon* codes [25] and *XOR* codes (e.g. RDP [7] and X-code [28]). Although there are storage systems based on both types of codes, the XOR codes outperform the others significantly [22] and form the basis of most recent storage systems [6, 14, 15, 23].

This paper addresses the issue of optimizing the encoding performance of XOR codes. XOR codes are described by equations that specify how the coding information is calculated from the data, and most implementations are constructed directly and naively from this specification. We call this naive implementation the *traditional* XOR-scheduling algorithm. To the best of our knowledge, the traditional algorithm is used not only in open source software such as Cleversafe's Information Dispersal implementation [14], Luby's Cauchy implementation [18], and Jerasure [20], but also in commercial software such as Netapp's RDP implementation [7]. However, the order of XOR operations is flexible and can impact the cache behavior of the codes significantly. Using this observation, we detail an algorithm named *DWG* (Data Words Guided) XOR-scheduling for performing these operations with their cache behavior in mind. *DWG* XOR-scheduling employs *loop transformation* [24] to optimize XOR operations. This optimization is based on understanding the semantics of erasure encoding and cannot be achieved by simple code transformations by a compiler.

We give a comprehensive demonstration of how *DWG* XOR-scheduling improves performance on a variety of XOR codes and processing environments. The bottom line is that *DWG* XOR-scheduling improves the raw performance of encoding by 21 to 45 percent on various machine architectures. Moreover, the improvement applies to all XOR codes and is therefore not code specific. Hence, *DWG* XOR-scheduling will be very useful for practical storage systems.

2 Nomenclature and Erasure Codes

A storage system is composed of an **array** of n disks, each of which is the same size. Of these n disks, k of them hold **data** and the remaining m hold **coding** information, often termed **parity**, which is calculated from the data. We label the data disks $D_0, \ldots, D_{k-1}$ and the coding disks $C_0, \ldots, C_{m-1}$. A typical system is pictured in Figure 1.

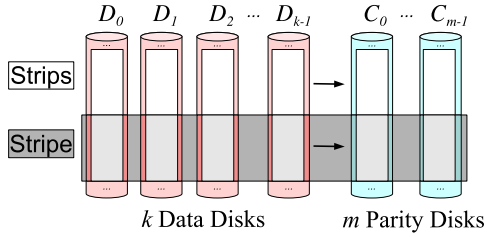


Figure 1. A typical storage system with erasure coding.

When encoding, one partitions each disk into **strips** of a fixed size. Each coding strip is encoded using one strip from each data disk, and the collection of $k + m$ strips that encode together is called a **stripe**. Thus, as in Figure 1, one may view each disk as a collection of strips, and one may view the entire system as a collection of stripes. Note that stripes are each encoded independently, and therefore if one desires to rotate the data and parity among the n disks for load balancing, one may do so by switching the disks' identities for each stripe.

Let us focus on a single stripe. Each XOR code has a parameter w, often termed the *word size* that further defines the code. This parameter is typically constrained by k and by the code. For example, for RDP [7], w must be less than or equal to k, and $w + 1$ must be a prime number. For the X-codes [28] and Liberation codes [21], w must be a prime number less than or equal to k.

Each strip is partitioned into exactly w contiguous regions of bytes, called *packets*, labeled $D_{i,0}, \ldots D_{i,w-1}$ and $C_{j,0}, \ldots C_{j,w-1}$ for data and coding drives D_i and C_j respectively. Each packet is the same size, called the *packet size*. Therefore, strip sizes are defined by the product of w and the packet size. An example of a stripe where $k = 4$, $m = 2$ and $w = 4$ is displayed in Figure 2.

For the purposes of defining a code, a packet size of one bit is convenient – coding bits are defined to be the XOR of collections of data bits. For example, in Figure 2, when the packet size is one, we can define the bit $C_{1,3}$ as being the XOR of bits $D_{3,0}$, $D_{2,1}$, $D_{1,2}$ and $D_{0,3}$, as it is in RDP [7]. However, for real implementations, packet sizes should be at least as big as a machine word, since CPUs can XOR two

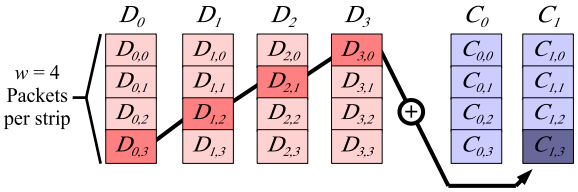


Figure 2. An example of one stripe where $k = 4$, $m = 2$ and $w = 4$.

words in one machine instruction. Moreover, to improve cache behavior, larger packet sizes are often preferable [22].

Codes are defined by specifying how each coding packet is constructed from the data packets. This may be done by listing equations, as in RDP [7], EVENODD [2] and X-code [28], or it may be done by employing a Generator Matrix. To describe a code with a Generator Matrix, let us assume that the packet size is one bit. Therefore each data and coding strip is a w-bit word and that their concatenation, which we label $(D|C)$ is a wn-bit word called the *codeword*. The Generator Matrix G has wk rows and wn columns and a specific format: $G = (I|H)$, where I is a $wk \times wk$ identity matrix. Encoding adheres to the following equation:

$$(D|C) = D * G = D * (I|H)$$

Thus, specifying the matrix H is sufficient to specify the code. Codes such as Liberation codes [21], Blaum-Roth codes [3] and Cauchy Reed-Solomon codes [5] are all specified in this manner.

Regardless of the specification technique, XOR codes boil down to lists of equations that construct coding packets as the XOR of collections of data packets. To be efficient, the total number of XOR operations should be minimized. Some codes, like RDP and X-code achieve a lower bound of $(k - 1)$ XOR operations per coding word, while others like Liberation codes, EVENODD and the STAR code are just above this lower bound [21, 22, 23].

2.1 Erasure Codes

We do not attempt to summarize all research concerning XOR codes. However, we detail the codes that are relevant to this paper. We only consider MDS codes. Cauchy Reed-Solomon codes [5] are general-purpose XOR codes that may be defined for any values of k and m. The word size w is constrained such that $w \geq 2^n$, and the resulting Generator Matrix is typically dense, resulting in a large number of XOR operations for encoding. Plank and Xu describe how to produce sparser matrices [23], and these represent the best performing general-purpose erasure codes.

When m is constrained to equal two, the storage system is a RAID-6 system, and the two coding drives are typically labeled P and Q. There are many XOR codes designed for RAID-6, and these outperform Cauchy Reed-Solomon codes significantly. As stated above, RDP [7] achieves optimal encoding performance, and Liberation codes [21], Blaum-Roth codes [3] and EVENODD [2] are slightly worse. The STAR code extends EVENODD for $m = 3$ and like EVENODD greatly outperforms Cauchy Reed-Solomon coding. The X-codes [28] and B-Codes [30] are RAID-6 codes that also achieve optimal encoding performance, but require the coding information to distributed evenly across all $(k+m)$ drives, and are therefore less flexible than the others.

Both Huang [12] and Hafner [10] provide techniques for grouping common XOR operations in certain codes to reduce their number. These techniques are especially effective for decoding Liberation and Blaum-Roth codes. In contrast to this work, we do not improve performance by reducing the number of XOR operations, but instead by improving how the order of XOR operations affects the cache.

Finally, in 2007, Plank released an open source erasure coding library called Jerasure [20] which implements both Reed-Solomon and XOR erasure codes. The XOR codes must use a Generator Matrix specification, with Cauchy Reed-Solomon, Liberation and Blaum-Roth codes included as basic codes. The XOR reduction technique of Hafner [10] is included to improve the performance of decoding. The library is implemented in C and has demonstrated excellent performance compared to other open-source erasure coding implementations [22].

3 CPU Cache

All modern processors have at least one CPU cache that lies between CPU and main memory. Its purpose is to bridge the performance gap between fast CPUs and relatively slow main memories [17]. Caches store recently referenced data, and when working effectively, reduce the number of accesses from CPU to main memory, each of which takes several instruction cycles and stalls the CPU. When an access to a piece of data is satisfied by the cache, it is called a cache *hit*; otherwise, it is called a cache *miss*. Many algorithms that optimize performance do so by reducing the number of cache misses.

There are three types of cache miss: *compulsory* miss, *capacity* miss, and *conflict* miss [11]. Compulsory misses happen when a piece of data is accessed for the first time. Capacity misses occur because of LRU replacement policies of limited-size caches, and conflict misses occur in A-way associative caches when two pieces of data map to the same cache location.

To reduce cache misses, an algorithm needs to have enough data locality in time, space, or both. *Temporal* locality is when the time period between two consecutive accesses to the same data is very short. *Spatial* locality is when the space difference between the data in a series of accesses is very small. Good temporal locality reduces capacity misses, and good spatial locality reduces both compulsory and conflict misses.

In section 2 above, we mention that large packet sizes are desirable. This is to reduce compulsory misses: when the first byte of a packet is read into the cache, its following bytes are also read into cache because of standard cache prefetch mechanisms. Then, when these bytes are used sequentially, their compulsory misses are avoided.

In this paper, we propose using *loop transformation* [17, 24, 26] to improve data locality when performing XOR operations. Specifically, we employ *loop interchange* and *loop fusion* techniques:

1. *Loop interchange*: for two adjacent loops in a loop nest, this transformation reverses the loop order. This technique is useful when an array is accessed in a loop nest. Reversing the loop order in a loop nest can reduce the stride of the accessed array data and hence improve *spatial* locality. Here, the stride is the memory distance of array elements accessed within consecutive inner loop iterations.

2. *Loop fusion*: for two consecutive loops which access some shared data, this transformation combines the two loops into one loop to improve *temporal* locality. If the two loops are not combined, the shared data might be moved out from CPU cache after the first loop is finished. Then, in the second loop, the access to the shared data will incur cache misses. By combining the two loops, these cache misses can be avoided.

4 XOR-Scheduling Algorithms

We motivate our scheduling algorithm with a toy example erasure code for $k = 2, m = 2$. In this code, $w = 1$, and our code is not an MDS code (since it is simply an example). We specify the code by assuming our packet size is one bit and listing the equations for the coding bits:

$$\begin{aligned} C_0 &= D_0 + D_1 \\ C_1 &= D_0 + D_1 \end{aligned}$$

Alternatively, we can specify our code with the following Generator Matrix:

$$G = (I|H) = \left(\begin{array}{cc|cc} 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 \end{array}\right)$$

Now, let us assume that we are implementing this code, and our packet size is two machine words. Let the two

words on data disk D_i be $d_{i,0}$ and $d_{i,1}$, and let the coding words be labeled similarly. The encoding of the entire system is straightforward, and shown in Table 1.

$c_{0,0} = d_{0,0} + d_{1,0}$
$c_{0,1} = d_{0,1} + d_{1,1}$
$c_{1,0} = d_{0,0} + d_{1,0}$
$c_{1,1} = d_{0,1} + d_{1,1}$

Table 1. Encoding the Toy Example when the packet size equals two.

4.1 Traditional XOR-scheduling Algorithm

Traditional XOR-scheduling performs the encoding in the order specified by the encoding equations or Generator Matrix / Data Vector product. In our example, the coding words $c_{0,0}$, $c_{0,1}$, $c_{1,0}$ and $c_{1,1}$ are calculated in that order. To ease the explanation, we are going to assume that each coding word starts with a value of zero, and its calculation requires two XORs to update it. It will be clear how to remove this assumption later.

The schedule of XOR operations generated by the traditional algorithm in our toy example is listed in Table 2. In the table, each row contains two columns. The second column is the XOR operation, and the first column is the ID for that XOR operation. Throughout our examples, the same XOR operation may appear in different positions in different scheduling algorithms, but its ID will remain constant.

ID	XOR
1	$c_{0,0} += d_{0,0}$
2	$c_{0,1} += d_{0,1}$
3	$c_{0,0} += d_{1,0}$
4	$c_{0,1} += d_{1,1}$
5	$c_{1,0} += d_{0,0}$
6	$c_{1,1} += d_{0,1}$
7	$c_{1,0} += d_{1,0}$
8	$c_{1,1} += d_{1,1}$

Table 2. The result of the traditional algorithm on the Toy Example.

The first characteristic of the traditional algorithm is that it processes coding packets one by one, to completion. In our toy erasure code, there are two coding packets: C_0 and C_1. In the traditional algorithm, the contents of C_1 will not be calculated until all of C_0 is finished. The second characteristic of the traditional algorithm is that when processing a coding packet, its associated data words are accessed in a strict order. For example, in Table 2, the execution sequence for coding packet C_0 is: $\{d_{0,0}, d_{0,1}, d_{1,0}, d_{1,1}\}$. In other words, when calculating C_0, none of D_1 is accessed until all of D_0 has been accessed.

A pseudocode description of the traditional XOR-scheduling algorithm is shown in Algorithm 1.

Algorithm 1 The Traditional Scheduling Algorithm

procedure $XOR\_scheduling()$

```
1: for each coding packet C_j do
2:   for each data packet D_i in the calculation of C_j do
3:     for x=0;x<packet size;x++ do
4:       output c_{j,x} += d_{i,x};
5:     end for
6:   end for
7: end for
```

Algorithm 1 shows that the traditional scheduling algorithm has good spatial locality because it accesses the words in a packet sequentially.

4.2 DWG XOR-scheduling Algorithm

The traditional XOR-scheduling algorithm follows the intuitive idea that coding words should be produced one by one. Instead, we can reorder the schedule so that it consumes *data* words one by one. Our new XOR-scheduling algorithm is based on this idea, and its characteristics are as follows:

1. The order of XOR operations is guided by the order of data words instead of coding words.
2. Each data word is used for all of its coding calculations before moving onto the next data word in the same packet.

Because of the first characteristic, the new algorithm is named *DWG* (Data Words Guided) XOR-scheduling. The result of *DWG* XOR-scheduling for the toy example is shown in Table 3.

The first characteristic requires that data packets are processed sequentially. As a result, in Table 3, all of the equations involving D_0 appear before those involving D_1. The second characteristic requires that each data word is used for all coding calculations before moving onto the next data word in the same packet. This can be seen in Table 3 where $d_{0,0}$ is used to calculate both $c_{0,0}$ and $c_{1,0}$ before $d_{0,1}$ is touched. A pseudocode description of this algorithm is shown in Algorithm 2.

ID	XOR
1	$c_{0,0} += d_{0,0}$
5	$c_{1,0} += d_{0,0}$
2	$c_{0,1} += d_{0,1}$
6	$c_{1,1} += d_{0,1}$
3	$c_{0,0} += d_{1,0}$
7	$c_{1,0} += d_{1,0}$
4	$c_{0,1} += d_{1,1}$
8	$c_{1,1} += d_{1,1}$

Table 3. The result of *DWG* XOR-scheduling on the Toy Example.

Algorithm 2 The *DWG* XOR-scheduling Algorithm

procedure $XOR\_scheduling()$

```
1: for each data packet D_i do
2:   for x=0;x<packet size;x++ do
3:     for each coding packet C_j that uses D_i do
4:       output c_{j,x} += d_{i,x};
5:     end for
6:   end for
7: end for
```

Algorithm 2 differs with Algorithm 1 in the ordering of the loops. Below are the reasons for why Algorithm 2 has better data locality than Algorithm 1.

1. Algorithm 2 first reverses the outermost two loops of Algorithm 1. This optimization is loop interchange, and it improves spatial locality.

2. After performing loop interchange, Algorithm 2 reverses the innermost two loops. This transformation looks like loop interchange, but instead it is loop fusion. This is clear from the source code. The innermost loop of Algorithm 2 shown in line 3 does not change the value of i and x, so it has good temporal locality by the same $d_{i,x}$ in line 4.

3. In Algorithm 2, the loop shown in line 3 still preserves good spatial locality to $c_{j,x}$ although some spatial locality may be lost. This is because in most reliable storage systems, k is bigger than m, and most data words will remain in the cache across iterations of the innermost loop. Thus, since $d_{i,x}$ and $c_{j,x}$ should remain in the cache across an iteration of the inner loop, $d_{i,x+1}$ and $c_{j,x+1}$ should remain there too.

4. Loop fusion can not be as effectively employed in Algorithm 1 when k is much larger than m. For example, a typical RAID-6 system will have $k \geq 10$, while $m = 2$. Thus, were we to switch the order of the two inner loops in Algorithm 1, we would be more likely to lose spatial benefits, because it is less likely that the data words will remain in the cache across iterations of the inner loop. Although we do not show the results, we did test loop fusion in Algorithm 1, and it did not improve performance.

We note that loop fusion may reduce some spatial locality in Algorithm 2. Usually, the benefit of temporal locality brought by loop fusion is greater than the loss of spatial locality, and it leads to better performance in Algorithm 2 than Algorithm 1. In some environments when using large packet sizes, the loss of spatial locality can be greater and result in worse performance. However, with a tunable packet size, Algorithm 2 always achieves the higher peak performance, which is important in practice and will be shown in Section 5.

Again, it is worth noting that employing loop interchange and loop fusion techniques in *DWG* XOR-scheduling needs to understand the semantics of encoding, and they are not simple code transformations that could be implemented by a compiler. In some respects, it is reminiscent of [17], which provides some examples of how manual optimization can greatly improve an algorithm's performance.

4.3 XOR-Scheduling Implementation Details

Clearly, generating and using schedules that are as detailed as those in Figures 2 and 3 take too much space. Fortunately, it is not necessary, since a list of each data packet's associated coding packets may be constructed simply from the packet's row of the Generator Matrix. Once that list is generated, it may be used for every word in the packet.

Another important detail of *DWG* XOR-scheduling is how to initialize coding packets. We cannot simply initialize them to zero, since that increases the number of XOR operations. Instead, we copy the first data packets that are used for each coding packet, instead of XOR-ing them. Implementationally, this is simple in Algorithm 1, where we may identify the first data packets at the beginning of Line 2. In Algorithm 2, it is more complex, since we need to know at Line 4 whether D_i is C_j's first data packet. Fortunately, all the codes that we address have regular structures which makes this determination straightforward.

A final detail involves the codes that have extra information in addition to their Generator Matrices – EVENODD, the STAR code and RDP. EVENODD employs a temporary packet, S that must be XOR'd with every packet in C_1. To handle this, both algorithms perform two passes. In the first pass, the data packets are used to calculate C_0, C_1 and S, and then a final pass XOR's S into the packets of C_1. The

Machine Name	CPU Speed	L1 Cache	L2 Cache	Cores	Bit	CPU Description	Memory
P4	3.0GHz	16KB	1MB	1	32-bit	Pentimum 4 (520)	1 GB
Pd	2.8GHz	2*16KB	2*1MB	2	64-bit	Pentimum Dual Core (D820)	1 GB
Pc2d	2.1GHz	2*32KB	3MB	2	64-bit	Pentimum Core 2 Duo (T8100)	2 GB
Pc2q	2.4GHz	4*32KB	2*4MB	4	64-bit	Pentimum Core 2 Quad (Q6600)	2 GB

Table 4. Details of the test platforms.

STAR code, which is an extension to EVENODD, is handled similarly. In RDP, all but one of the packets in C_1 are calculated using packets in C_0 in addition to data packets. For that reason, we also perform two passes in RDP: a first one that calculates C_0 and C_1 without the C_0 packets, and a second one that XOR's the C_0 packets into C_1.

5 Performance Evaluation

To compare the two XOR-scheduling algorithms, we have conducted experiments that apply both algorithms to many popular XOR-based erasure codes: the Liberation codes [21], EVENODD [2], RDP [7], X-code [28] and the STAR code [13]. All but the STAR code are RAID-6 codes. Since CPU cache behavior is complicated, we have run our experiments on various machines to get a comprehensive view of *DWG* XOR-scheduling. There are four platforms used in our experiments, which all use Intel processors ranging from low-end to high-end. The detailed information on these platforms are shown in Table 4.

All platforms run Linux, and the 64-bit machines run the 64-bit version of Linux. Since 64-bit machines perform XOR operations on 64-bit words, their encoding performance is as much as a factor of two faster than their 32-bit counterparts [22]. Our code is written in C and compiled using **gcc** with the **-O2** and **-O3** optimization flags set respectively. **-O2** is recommended for most applications, while **-O3** is the highest level of optimization [8]. Since we did not observe any significant performance difference between **-O2** and **-O3**, we only present the performance results for **-O2**. Matching open source implementations, the code runs only on one thread, and thus does not take advantage of multiple cores.

For the Liberation codes, we use the Jerasure open source coding library [20] as the implementation for the traditional XOR-scheduling algorithm, as that is exactly what Jerasure implements. For all other codes and for implementing *DWG* XOR-scheduling, we wrote our own implementations, starting with Jerasure as the base code.

5.1 Experiment Setup

We set up an experimental framework to evaluate XOR-scheduling algorithms' performance. In our framework, all operations are performed in memory with no disk I/O involved. The reason is that employing the disk makes it very difficult to assess encoding performance accurately [22].

The size of total input user data is 1 GB, which we assume will be shared among k data devices and encoded onto m coding devices. We first allocate 20 MB in main memory as a data buffer. Then, each time we randomly pick up an area in this data buffer as a coding buffer, and proceed in phases until all the 1 GB user data has been encoded. To simulate reading the data, we fill the buffer randomly.

We evaluate encoding performance using a metric of *Encoding Bandwidth*, calculated with the equation below:

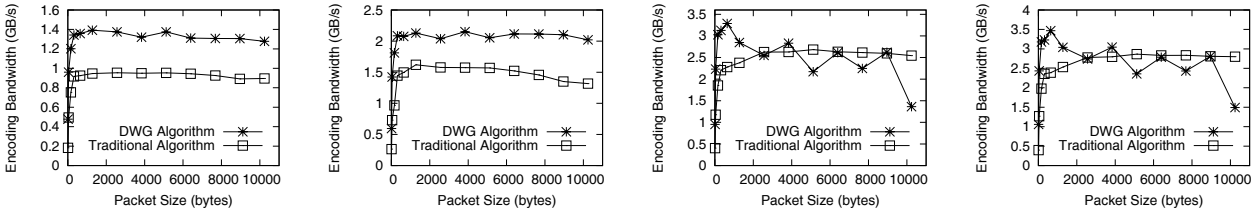
$$\text{Encoding Bandwidth} = \frac{\text{Encoding Time}}{\text{Total Input Data Size}}$$

We use Encoding Bandwidth so that the performance evaluation is not dependent on the size of the data. It represents how quickly one can turn data into coding information with a given code, algorithm and machine. Since all but one of the codes evaluated are RAID-6 codes, their performance may be evaluated directly. The STAR code will encode with a lower bandwidth, since it creates three drives' worth of coding information rather than two. The encoding time is calculated using the **gettimeofday()** system call. Each data point is the average of 50 test runs. The ratio of the standard deviation to the average value is at most 5%.

5.2 Experiments on the Liberation Codes

5.2.1 Parameters k=w=11, m=2

We first focus on a performance comparison of the Liberation codes. We do this because it is the one code for which we have an open source implementation. The parameters of our first experiment are $k = 11$, $m = 2$ and $w = 11$, as these represent a typical RAID-6 system. We modify the packet size, as [22] has demonstrated that varying packet sizes can impact performance.



(a) P4, $k = w = 11$ (b) Pd, $k = w = 11$ (c) Pc2d, $k = w = 11$ (d) Pc2q, $k = w = 11$

Figure 3. Encoding performance of Liberation codes, $k = w = 11$ and $m = 2$.

Figure 3(a) shows the performance comparison on machine P4. Figure 3(a) allows us to make the following observations:

1. As packet sizes increase from a small value, the performance of both scheduling algorithms improves significantly.
2. For all packet sizes, *DWG* XOR-scheduling achieves much better encoding performance than the traditional one.
3. The peak performance of *DWG* XOR-scheduling is much higher than that of the traditional one.
4. *DWG* XOR-scheduling achieves its peak encoding performance with a relatively small packet size.

The first observation mirrors the observations in [22] which reflects greater spatial locality with larger packet sizes. The reason for the second and third observations is *DWG* XOR-scheduling has better temporal locality than the traditional algorithm, matching our analysis in Section 4.2. The last observation is important and will recur in the data below. The traditional algorithm generates completed coding words quickly and sequentially. *DWG* XOR-scheduling in contrast only completes generating coding data at the end of each stripe. Therefore, we may prefer smaller packets with this algorithm, and it is important that it performs well with relatively small packet sizes.

Figure 3(b) displays the results on the Pd machine. The four above observations on P4 apply also to Pd. The algorithms encode faster though, since Pd is a 64-bit machine.

Figure 3(c) shows the comparison results on machine Pc2d. Although the machine has a slower processor than the others, it has larger caches and achieves better peak performance. We also see a new trend on this machine – after reaching the peak performance with a small packet size, the performance of *DWG* XOR-scheduling becomes unstable, and at some points it performs worse than the traditional algorithm. A possible reason is that the data locality in *DWG* XOR-scheduling is more sensitive to the packet size than the traditional algorithm. This platform, as shown in Table 4, has a dual core processor, and the L2 cache is shared by two cores. This property is different from previous platforms, where the L2 cache is dedicated for one core. Therefore, the L2 cache may have more effect on *DWG* XOR-scheduling since *DWG* XOR-scheduling exploits both temporal and spatial locality while the traditional algorithm only uses spatial locality. Finally, we still see that the two most important observations hold: 1) the peak performance of *DWG* XOR-scheduling is much higher than that of the traditional one; 2) the peak performance is obtained with small packet size.

Figure 3(d) shows the results on machine Pc2q. The performance of Pc2q is better than the others, but the trends are very similar to Pc2d, and the observations for Pc2d are also valid for Pc2q.

The peak performance of two scheduling algorithms on the four machines are summarized in Table 5. It clearly displays that *DWG* XOR-scheduling achieves significant performance improvement, from 21% to 45%.

Machine	DWG	Traditional	Improvement
P4	1.392 GB/s	0.955 GB/s	45%
Pd	2.129	1.619	31%
Pc2d	3.287	2.684	22%
Pc2q	3.467	2.865	21%

Table 5. Comparison of peak encoding performance of Liberation codes for $k = w = 11$ and $m = 2$.

5.2.2 Modifying k and w

In the Liberation codes, the number of coding devices is fixed at two. However, systems may range in size from small k to large.

To observe potential sensitivity to the number of data devices k and the number of packets per stripe w, in Figure 4 we display the encoding performance of the Liberation codes for $k = w = 5$ and $k = w = 17$. In the top row of graphs, k and w equal 5, and in the lower row they

equal 17. The results mirror the results for $k = 11$, and the observations that held for $k = 11$ hold for the smaller and larger values of k. Thus, *DWG* XOR-scheduling is stable for this code among this range of parameters.

5.3 Experiments on Other Erasure Codes

To compare the algorithms on other codes, we tested EVENODD, RDP, X-code and the STAR code in our framework. Except for the STAR code, the other three codes are RAID-6 codes. EVENODD and RDP have efficient decoding algorithms, while RDP has better encoding performance and EVENODD has better update performance [23]. X-code has good encoding/decoding/update performance; however, it is a vertical code and has limitations on the choice of the value of k. Lastly, the STAR code can tolerate up to 3 disk failures. These various codes impose different constraints on k and w, so we selected values that would match most closely with the Liberation codes example. The values are summarized in Table 6.

Code	k	w	m
EVENODD	11	10	2
RDP	10	10	2
X-code	11	11	2
STAR	11	10	3

Table 6. Encoding parameters of various codes.

The results are in Figure 5. In terms of peak performance, the codes match expectations. When $k = w$, the Liberation codes'performance is theoretically identical to EVENODD [21], and this is reflected in the results. RDP and X-code should encode the fastest, and they do. The STAR code's performance is worse than the others because it encodes an extra coding device.

In terms of the trends, the codes' performance matches the Liberation codes, and all the observations for the Liberation codes hold for these codes as well. The results show that the algorithm is applicable to a wide variety of codes.

5.4 Encoding Performance of RDP

In [7], Corbett provides encoding performance of RDP Code, and it is the best reported result of RDP Code in the literature. The platform used in [7] contains two Pentium 4 CPUs of 2.8GHz of which one CPU is dedicated for RDP encoding. They do not report the cache sizes. The most similar platform in our tests is machine P4, but P4 only has one CPU. The performance comparison of *DWG* XOR-scheduling with their reported performance is given in Table 7.

Machine	DWG	Reported in [7]
P4	1.336 GB/s	1.55 GB/s

Table 7. Encoding performance of RDP for $k = w = 10$ and $m = 2$.

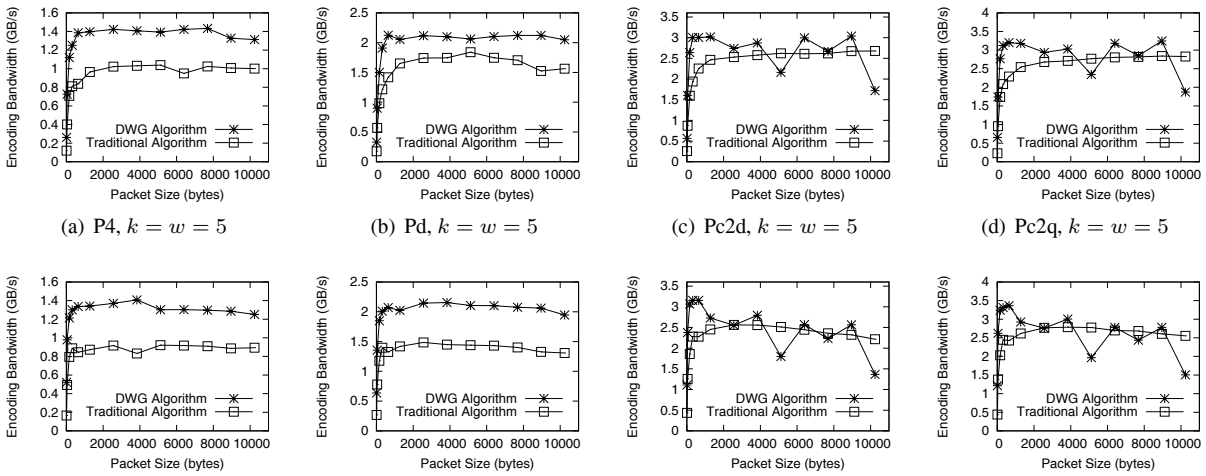
Table 7 shows that the encoding performance achieved by *DWG* XOR-scheduling is a little worse than their reported performance. Without knowing the exact details, the reasons can be manyfold. However, a simple conclusion to draw is that with *DWG* XOR-scheduling, our implementation can achieve encoding performance that is comparable to an implementation used in commercial products.

6 Conclusions

This paper proposes a new algorithm named *DWG* XOR-scheduling to improve the encoding performance of XOR-based erasure codes. For these erasure codes, the encoding performance is determined by two primary factors: the number of XOR operations and the cache behavior. *DWG* XOR-scheduling is able to efficiently utilize CPU cache and thus achieves much better encoding performance than the traditional algorithm. In a performance evaluation on some widely known erasure codes on a variety of platforms, we show that the encoding performance obtained by *DWG* XOR-scheduling significantly outperforms that of the traditional algorithm.

Two clear areas of future work are to incorporate *DWG* XOR-scheduling into **Jerasure** so that those who use this open source tool may take advantage of our work, and to assess the impact of this technique on decoding. Since decoding is a more complex activity than encoding and occurs more rarely, we have not yet performed this assessment. It will be a necessary step to providing a complete evaluation of the scheduling algorithm.

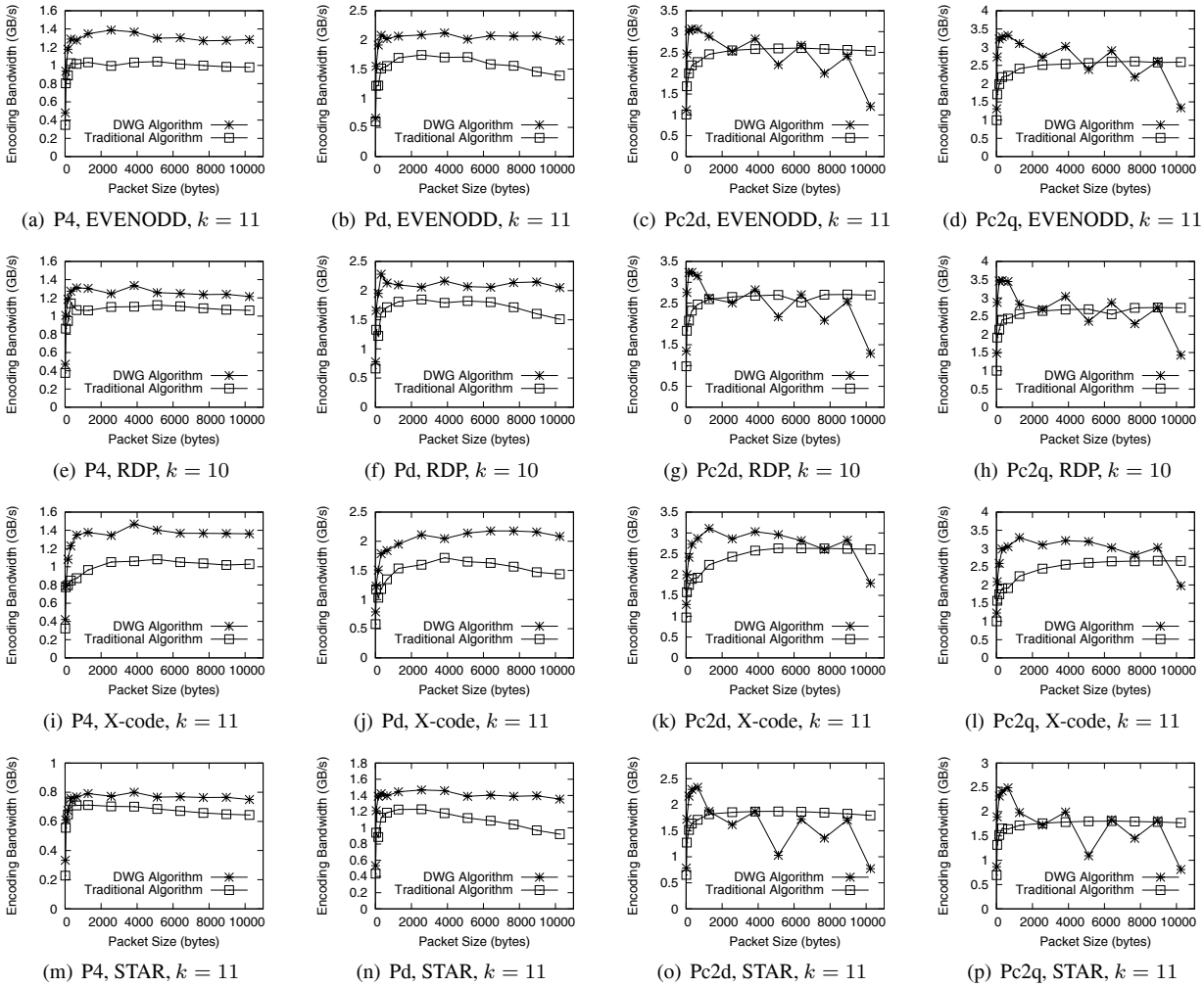
Additional future work is to put *DWG* XOR-scheduling into a real storage system. We plan to implement a reliable storage system [29] and use various scheduling algorithms in it to find how *DWG* XOR-scheduling can improve this system's performance. Another future work is to find similar optimization approaches for Reed-Solomon Codes. Although they do not perform as well as the XOR based codes in this paper, Reed-Solomon Codes are used widely not only in storage systems but also in network systems. Thus the performance improvements of our techniques will have wider impact.



(a) P4, $k = w = 5$ (b) Pd, $k = w = 5$ (c) Pc2d, $k = w = 5$ (d) Pc2q, $k = w = 5$

(e) P4, $k = w = 17$ (f) Pd, $k = w = 17$ (g) Pc2d, $k = w = 17$ (h) Pc2q, $k = w = 17$

Figure 4. Encoding performance of Liberation codes, varying k and w from 5 to 17.



(a) P4, EVENODD, $k = 11$ (b) Pd, EVENODD, $k = 11$ (c) Pc2d, EVENODD, $k = 11$ (d) Pc2q, EVENODD, $k = 11$

(e) P4, RDP, $k = 10$ (f) Pd, RDP, $k = 10$ (g) Pc2d, RDP, $k = 10$ (h) Pc2q, RDP, $k = 10$

(i) P4, X-code, $k = 11$ (j) Pd, X-code, $k = 11$ (k) Pc2d, X-code, $k = 11$ (l) Pc2q, X-code, $k = 11$

(m) P4, STAR, $k = 11$ (n) Pd, STAR, $k = 11$ (o) Pc2d, STAR, $k = 11$ (p) Pc2q, STAR, $k = 11$

Figure 5. Encoding performance of various erasure codes.

Acknowledgments

We would like to thank our shepherd, Jay Wylie, and the anonymous reviewers for their constructive comments and suggestions. This research is supported by National Science Foundation under grants IIS-0541527 and CNS-0615221.

References

[1] Allmydata. Unlimited Online Backup, Storage, and Sharing, 2008. http://allmydata.com.

[2] M. Blaum, J. Brady, J. Bruck, and J. Menon. EVENODD: An Efficient Scheme for Tolerating Double Disk Failures in RAID Architectures. *IEEE Transactions on Computers*, 44 (2):192–202, February 1995.

[3] M. Blaum and R. M. Roth. New Array Codes for Multiple Phased Burst Correction. *IEEE Transactions on Information Theory*, 39 (1):66–77, January 1993.

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

[5] J. Blomer, M. Kalfane, M. Karpinski, R. Karp, M. Luby, and D. Zuckerman. An XOR-based Erasure-Resilient Coding Scheme. *Technical Report TR-95-048, International Computer Science Institute*, August 1995.

[6] V. Bohossian, C. C. Fan, P. S. LeMahieu, M. D. Riedel, J. Bruck, and L. Xu. Computing in the RAIN: A Reliable Array of Independent Nodes. *IEEE Transaction on Parallel and Distributed Systems*, 12(2):99–114, 2001.

[7] P. Corbett, B. English, A. Goel, T. Grcanac, S. Kleiman, J. Leong, and S. Sankar. Row-Diagonal Parity for Double Disk Failure Correction. In *FAST '04: Proc. of the 3rd USENIX Conference on File and Storage Technologies*, March 2004.

[8] gentoo wiki. http://en.gentoo-wiki.com/wiki/CFLAGS.

[9] J. L. Hafner. WEAVER Codes: Highly Fault Tolerant Erasure Codes for Storage Systems. In *FAST '05: Proc. of the 4th USENIX Conference on File and Storage Technologies*, December 2005.

[10] J. L. Hafner, V. Deenadhayalan, KK Rao, and J. A. Tomlin. Matrix Methods for Lost Data Reconstruction in Erasure Codes. In *FAST '05: Proc. of the 4th USENIX Conference on File and Storage Technologies*, December 2005.

[11] J. L. Hennessy, D. A. Patterson, and D. Goldberg. *Computer Architecture: A Quantitative Approach*, chapter Appendiex C. Morgan Kaufmann, May 2002.

[12] C. Huang, J. Li, and M. Chen. On Optimizing XOR-Based Codes for Fault-Tolerant Storage Applications. In *ITW '07: Proc. of IEEE Information Theory Workshop*, September 2007.

[13] C. Huang and L. Xu. STAR: An Efficient Coding Scheme for Correcting Triple Storage Node Failures. In *FAST '05: Proc. of the 4th USENIX Conference on File and Storage Technologies*, December 2005.

[14] CLEVERSAFE INC. Cleversafe Dispersed Storage, 2008. http://www.cleversafe.org/downloads.

[15] N. Joukov, A. M. Krishnakumar, C. Patti, A. Rai, S. Satnur, A. Traeger, and E. Zadok. RAIF: Redundant Array of Independent Filesystems. In *MSST '07: Proc. of the 24th IEEE Conference on Mass Storage Systems and Technologies*, pages 199–212, September 2007.

[16] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, C. Wells, and B. Zhao. OceanStore: An Architecture for Global-scale Persistent Storage. In *ASPLOS '00: Proc. of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, December 2000.

[17] A. R. Lebeck and D. A. Wood. Cache Profiling and the SPEC Benchmarks: A Case Study. *IEEE Computer*, 27 (10):15–26, 1994.

[18] M. Luby. Code for Cauchy Reed-Solomon Coding, 1997. http://www.icsi.berkeley.edu/ luby/cauchy.tar.uu.

[19] B. Nisbet. FAS storage systems: Laying the foundation for application availability. *Network Appliance white paper: http://www.netapp.com/us/library/analyst-reports/ar1056.html*, February 2008.

[20] J. S. Plank. Jerasure: A Library in C/C++ Facilitating Erasure Coding for Storage Applications. *Tech. Rep. CS-07-603, University of Tennessee*, September 2007.

[21] J. S. Plank. The RAID-6 Liberation Codes. In *FAST '08: Proc. of the 6th Usenix Conference on File and Storage Technologies*, February 2008.

[22] 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 '09: Proc. of the 7th Usenix Conference on File and Storage Technologies*, February 2009.

[23] J. S. Plank and L. Xu. Optimizing Cauchy Reed Solomon Codes for Fault-Tolerant Network Storage Applications. In *NCA '06: Proc. of the 5th IEEE International Symposium on Network Computing and Applications*, July 2006.

[24] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann, October 2001.

[25] I. S. Reed and G. Solomon. Polynomial Codes over Certain Finite Fields. *Journal of the Society for Industrial and Applied Mathematics*, 8 (10):300–304, 1960.

[26] J. Warren. A Hierarchical Basis for Reordering Transformations. In *POPL '84: Proc. of the 11th ACM SIGACT-SIGPLAN symposium on Principles of Programming Languages*, January 1984.

[27] J. J. Wylie and R. Swaminathan. Determining Fault Tolerance of XOR-Based Erasure Codes Efficiently. In *DSN '07: Proc. of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, June 2007.

[28] L. Xu. X-Code: MDS Array Codes with Optimal Encoding. *IEEE Transactions on Information Theory*, 45 (1):272–276, January 1999.

[29] L. Xu. Hydra: A Platform for Survivable and Secure Data Storage Systems. In *StorageSS '05: International Workshop on Storage Security and Survivability*, November 2005.

[30] L. Xu and J. Bruck. Low Density MDS Code and Factors of Complete Graphs. *IEEE Transactions on Information Theory*, 45:1817–1826, 1999.