Efficient Encoding Schedules for XOR-Based Erasure Codes

Jianqiang Luo, Mochan Shrestha, Lihao Xu, and James S. Plank

Abstract—In data 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 parity 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. We observed that CPU cache efficiency has great impact on the encoding performance and proposed several encoding scheduling algorithms to optimize the use of cache memory. We call the technique XOR-scheduling and demonstrate how it applies to a wide variety of existing erasure codes. To illustrate the generality of this technique, we have conducted a performance evaluation of scheduling these codes on a variety of platforms and shown that XOR-scheduling significantly improves upon the conventional approach. Hence, we believe that XOR-scheduling has great potential to have wide impact in practical storage systems.

Index Terms—Erasure correcting codes, storage systems

1 Introduction

As the amount of data increases exponentially in large distributed 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 [1], RAIF [2], and RAIN [3], and companies like Network Appliance [4], HP [5], IBM [6], Cleversafe [7], employing erasure codes such as RDP [8], the B-Code [9] and Reed-Solomon Codes [10], [11].

In an erasure coded system, a total of $n = k + m$ storage devices are employed, of which k hold data and m hold parity. The act of *encoding* calculates the parity from the data, and *decoding* reconstructs the data from surviving storage devices following one or more failures. Storage systems typically employ *Maximum Distance Separable (MDS)* codes [12], which ensure that the data can always be reconstructed as long as there are at least k storage devices that survive the failures.

Encoding is an operation performed constantly as new data is written to the system. Because encoding operation is a highly frequent operation, its performance is crucial to the overall system performance. As flash-based solid-state drive (SSD) rapidly enters the enterprise data storage [13], improving encoding performance becomes more important. That is because I/O cost is much lower than using spinning disks, and thus CPU resource becomes more valuable. There are two classes of MDS codes—*Reed-Solomon* codes [11] and *XOR* codes (e.g. RDP [8] and X-code [14]). Although some storage systems may use Reed-Solomon codes for flexibility, the XOR codes outperform the others significantly [15] and form the basis of most recent storage systems [3], [7], [2], [16].

This paper addresses the issue of optimizing the encoding performance for XOR-based erasure codes. For an XOR code, its parity is calculated from the data by bitwise exclusive-or operation. We observed that due to exclusive-or is a fast operation, encoding for XOR codes are indeed memory-bound. One way to improve the cache efficiency is to make the XOR-operations more CPU-cache friendly. This is the focus of this paper. We first define that an *XOR-Scheduling algorithm* is an algorithm that performs a number of XOR operations in a certain order.

We found that most existing implementations are constructed directly and intuitively from the code specification. We call this straightforward implementation the conventional XOR-scheduling algorithm. The conventional algorithm is used in open source software such as Cleversafe's Information Dispersal implementation [7], Luby's Cauchy implementation [17], and Jerasure [18]. However, the order of XOR operations is flexible and can impact cache memory behavior significantly. Using this observation, we analyzed the encoding process of XOR codes and proposed three new XOR-scheduling algorithms. These algorithms choose different scheduling orders for the same XOR operations, and then achieve different encoding performance.

In this paper, we focus on raw encoding performance just as in [15], [8]. We give a comprehensive demonstration of how two proposed new XOR-scheduling algorithms improve encoding performance on a variety of XOR codes and processing environments. The performance evaluation covers four platforms and five erasure codes. Finally, we found that two of the new algorithms, called Data Words Guided (DWG) and Data Packets Guided (DPG), outperform the others. *DWG*

• *J. Luo, M. Shrestha, and L. Xu are with the Department of Computer Science, Wayne State University, Detroit, MI 48202.*
E-mail: {jianqiang, mochan, lihao}@cs.wayne.edu.
• *J. S. Plank is with the Department of Electrical Engineering and Computer Science, University of Tennessee, Knoxville, TN 37996.*
E-mail: plank@cs.utk.edu.

Manuscript received 23 Aug. 2012; revised 31 Dec. 2012; accepted 17 Jan. 2013. Date of publication 30 May 2013; date of current version 07 Aug. 2014.
Recommended for acceptance by C. Bolchini.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TC.2013.23

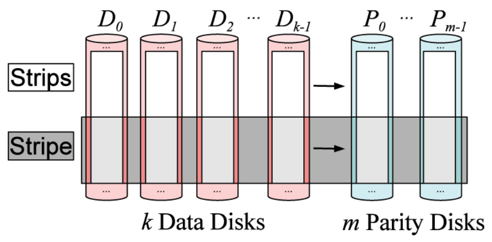


Fig. 1. A typical storage system with erasure coding.

improves the raw performance of encoding by 23% to 36% on various platforms. Moreover, the improvement applies to all tested XOR codes and is therefore not code specific. To further understand the performance of the XOR-scheduling algorithms, instead of simulations or simplistic modeling, we perform a profiling analysis of our tests with the VTune Performance Analyzer [19] to illuminate the different cache effects that indeed impact these scheduling algorithms. Finally, by tuning different compiler optimization flags, we observed that the performance improvement obtained by the new algorithms cannot be simply achieved by compiler optimization.

The rest of the paper is organized as follows. Section 2 discusses the related background of erasure codes. Section 3 briefly describes how CPU cache impacts encoding performance. Section 4 presents this paper's main result: four different XOR-scheduling algorithms. The performance evaluation of the XOR-scheduling algorithms is provided in Section 5. Section 6 concludes the paper.

2 Background

A storage system is composed of an array of n disks, each of which is the same size. Of these n disks, k hold data information 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 parity disks $P_0, \ldots, P_{m-1}$. A typical system is pictured in Fig. 1.

When encoding, one partitions each disk into strips of a fixed size. Each strip for a parity disk is encoded using one strip from each data disk, and the collection of $k+m$ strips is called a *stripe*. Thus, as in Fig. 1, one may view each disk as a collection of strips, and one may view the entire system as a collection of stripes. The 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 that further defines the code. This parameter is typically constrained by k and by the code. For example, for RDP [8], $w+1$ must be a prime number, while for the X-codes [14] and Liberation codes [20], w must be a prime number.

Each strip is partitioned into exactly w contiguous regions of bytes, called *packets*, labeled $D_{i,0}, \ldots D_{i,w-1}$ and $P_{j,0}, \ldots P_{j,w-1}$ for data and parity disks D_i and P_j respectively. Each packet is the same size, i.e., containing the same number of bits, called the *packet size*. Strip sizes are therefore 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 Fig. 2.

For the purpose of defining a code, a packet size of one bit is convenient–parity bits are defined to be the XOR of collections

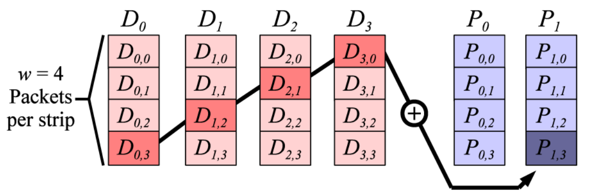


Fig. 2. An example of one stripe where $k=4$, $m=2$ and $w=4$.

of data bits. For example, in Fig. 2, when the packet size is one, we can define the bit $P_{1,3}$ as being the XOR of bits $D_{0,3}$, $D_{1,2}$, $D_{2,1}$ and $D_{3,0}$, as it is in RDP [8]. However, for real implementations, packet sizes should be at least as big as a machine word, since CPUs can XOR two words in one machine instruction. Moreover, to improve CPU cache behavior, larger packet sizes are often preferable [15].

Codes are defined by specifying how each parity packet is constructed from the data packets. This may be done by listing equations, as in RDP [8], EVENODD [21], and X-code [14], or it may be done by employing a generator matrix [22]. To describe a code with a generator matrix, let us assume that the packet size is one bit. Therefore each data and parity strip is a w-bit word and their concatenation, which we label $(D|P)$, 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:

$$(D|P) = D * G = D * (I|H).$$

Thus, specifying the matrix H is sufficient to specify the code. Codes such as Liberation codes [20], Blaum-Roth codes [23] and Cauchy Reed-Solomon codes [10] are all specified in this manner.

Regardless of the specification technique, XOR codes boil down to lists of equations that construct parity 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 parity word, while others like Liberation codes, EVENODD and the STAR code are just above this lower bound [20], [15], [16].

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 focus on MDS codes. Cauchy Reed-Solomon codes [10] are general-purpose XOR codes that may be defined for any values of k and m. w is constrained such that $2^w \geq n$, 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 [16], 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. There are many XOR codes designed for RAID-6, and these outperform Cauchy Reed-Solomon codes significantly. As stated above, RDP [8] achieves optimal encoding performance, and Liberation codes [20], Blaum-Roth codes [23] and EVENODD [21] are slightly worse. The STAR code extends EVENODD for $m=3$ and like EVENODD greatly outperforms Cauchy Reed-Solomon coding. The X-codes [14] and B-Codes [9] are RAID-6 codes that also achieve optimal encoding performance, but require the parity information to be distributed evenly across all $(k+m)$ storage devices, and are therefore less flexible than the others.

Both Huang [24] and Hafner [25] 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 cache behavior.

Finally, in 2007, Plank released an open source erasure coding library called Jerasure [18] 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 [25] 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 [15].

3 CPU Cache

In this section, we will first introduce the general CPU cache concepts related to Intel CPUs, which are used in our experiments, and then show the impact of CPU cache efficiency.

3.1 Introduction

Generally, all modern processors have CPU cache that lies between CPU and main memory. Caches store recently referenced data. Its purpose is to bridge the performance gap between fast CPUs and relatively slow main memories [26]. The latency of cache access is only a few to tens of CPU cycles, while the latency of memory access can be hundreds of cycles [27]. Therefore, when working effectively, cache can reduce the number of accesses from CPU to main memory and improve performance.

Data transferred from CPU and memory is fixed size, called *cache line*. A typical cache line is 64 bytes. There are two different types of caches: instruction cache which stores executable instructions, and data cache which stores data. Cache performance relates to cache size. Larger cache can store more data but has longer access latency. Due to it, data cache is organized in a hierarchy of multiple cache levels, with smaller faster caches backed up by larger slower caches. For example, Intel Core i7-3770 K has 128 K of L1 data cache, 1 MB of L2 cache, and 8 MB of L3 cache.

When a processor needs to access a piece of data, it first checks if the data is in cache or not. If the data is already in cache, it is called *cache hit*; otherwise, the data has to be loaded from memory and stored in cache, called *cache miss*. There are three basic types of cache miss: *compulsory* miss, *capacity* miss, and *conflict* miss [28], [26], [29], [30]. Compulsory miss happens when a piece of data is accessed for the first time. Cache prefetching would help this case. Capacity miss occurs due to limited-size caches. This miss can be reduced if a larger cache size is used or a CPU-cache friendly algorithm is implemented. Last, conflict miss happens when two pieces of data map to the same cache line because of non-fully associative cache. If data access is not aligned, the probability of conflict is likely to be increased [27].

To increase cache hits or reduce cache misses, an algorithm needs to be CPU-cache friendly, i.e., showing sufficient data locality in time, space, or both. *Temporal* locality indicates the time period between two consecutive accesses to the same data is very short. Hence, after a piece of data is loaded into cache, the data can be repeatedly accessed in cache. *Spatial* locality means the memory distance between the data in a series of accesses is very small. As a result, after a piece of data is loaded, its neighbor data is also pre-fetched into the cache, and then the data access to the neighbor data can be completed in cache. Typically, good temporal locality can reduce capacity misses, and good spatial locality is effective to reduce both compulsory and conflict misses.

3.2 The Impact of CPU Cache

We observed that the impact of CPU cache on encoding performance for XOR-based erasure codes is significant.

We first analyze the encoding performance for the worst case, i.e., CPU cache hit ratio is 0%. Consequentially, during the encoding operation, all data access has to go through memory. Now we use EVENODD as an example for the detailed analysis. EVENODD is a RAID-6 code, which contains p data columns and 2 parity columns. The row number is $p-1$. Hence, a codeword contains $(p-1)*p$ packets of user data. Suppose the packet size is 64-bits. It is easy to derive that the number of exclusive-or operations for a codeword is $2p^2-2p-1$. For simplicity, we use $2p^2-2p$. As each exclusive-or computation involves two memory read and one memory write, and all access has to go to memory, the number of bytes for memory access will be $(2p^2-2p)*3*8$. To evaluate encoding performance, we define *encoding rate* as the speed of encoding user data. Denote the maximum theoretical memory bandwidth as Mem. By testing $memcpy$, we observed that 75% of the Mem bandwidth can be practically achieved. Now we can calculate the encoding rate for this case as below:

$$\frac{(p-1)*p*8}{((2p^2-2p)*3*8)/(Mem*3/4)}=\frac{1}{6}*Mem*3/4.$$

Next we consider another extreme case, i.e., CPU cache hit ratio is 100%. The effect is that after a piece of data is loaded from memory to cache, the data can be always accessed from the cache; similarly, if a piece of data in memory needs to be updated, it will be first updated in cache, and then stored to memory after the last update. Different from the worst case of cache hit ratio, in this setting, the amount of memory access is determined by the amount of unique data access, rather than the amount of exclusive-ors. This greatly improves the performance. The calculated encoding rate will be as below:

$$\frac{(p-1)*p*8}{((p-1)*(p+2)*8)/(Mem*3/4)}=\frac{p}{(p+2)}*Mem*3/4.$$

We simulated the above analysis on a real platform and calculated the encoding rate. The platform for the simulation uses a CPU with 3.5 GHz, so that CPU would not be the performance bottleneck. For EVENODD, we chose $p=11$.

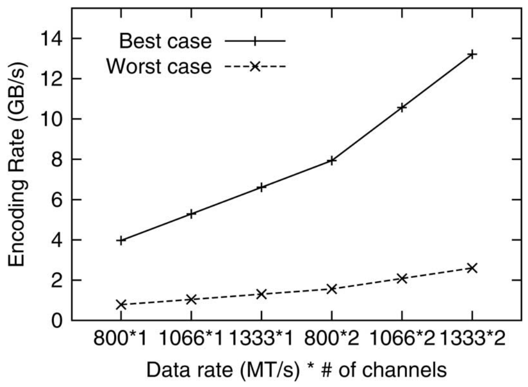


Fig. 3. Encoding rate with different cache efficiency.

The tested memory types include the mostly common ones, such as 800MT/s, 1066MT/s, and 1333MT/s data rate. Each memory type is tested against both single channel and dual channel mode. As memory width is 64-bits, i.e., 8 bytes, the formula to compute memory bandwidth is:

$$Mem = \text{data rate} * \# \text{ of channels} * 8 \quad (\text{MB/s}).$$

The simulation results are plotted in Fig. 3. We can have two observations from the figure. First, the encoding rate differs dramatically for the two cases. For all tested memory configurations, the encoding rate for the best case is five times as that for the worst case. The results indicate that the cache efficiency greatly impacts the encoding performance, and then one natural way to improve the encoding performance is to use CPU cache more efficiently. Second, the performance trend across different memory bandwidth shows that encoding is actually a memory-bound operation. Hence, using higher bandwidth memory is also an effective way to improve the performance.

4 XOR-Scheduling Algorithms

As shown in the above section, for an XOR-based erasure codes, CPU cache efficiency greatly impacts the encoding performance. Naturally, we need to make the encoding algorithm more CPU-cache friendly. Here, we define an XOR-Scheduling algorithm as the following:

Definition 1. *An XOR-Scheduling algorithm is an algorithm that performs a number of XOR operations in a certain order.*

As we would demonstrate in Section 5, for the same amount of XOR operations, different XOR-scheduling algorithms provide significantly different encoding performance.

4.1 A Toy Example Code

We motivate our scheduling algorithms with a toy example erasure code for $k = 2, m = 2$, and $w = 2$. Although this code is simply a toy example, it is close to the EVENODD code. The toy code is described by Fig. 4. The code has two data columns: D_0 and D_1, and two parity columns: P_0 and P_1. The middle part of Fig. 4 displays the code construction in terms of packets. Now, assume when implementing this code, the packet size is two, i.e., each packet contains two machine words. Let us label the t^{th} word on one data packet $D_{[j,i]}$ (at data column j and row i) as $D_{[j,i],t}$, similarly for the parity words. Then, the right part of the figure lists all the needed exclusive-or (XOR) operations based on words. Here, notation '+' denotes XOR operation.

Following the data layout in practice, each column of the toy code is one byte array. The benefit is all the information of one column can be read/written to from/to a storage device by a single read/write request. As memory layout is related to the encoding performance, we use the same layout in this paper.

Throughout this section, we will stick to the toy example. Using this toy example will make it easy to demonstrate the XOR-scheduling algorithms. In the performance evaluation section, we will use real erasure codes.

4.2 Conventional XOR-Scheduling Algorithm

Conventional XOR-scheduling refers to the algorithms that are intuitively implemented according to an erasure code's specification. The conventional algorithm is used in several open source software [7], [17], [18]. For the toy example, the schedule of XOR operations generated by the conventional algorithm is described in Fig. 5.

Note that in the toy example, to ease the explanation, we are going to assume that each parity 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 in Section 4.6.

In Fig. 5, the left part lists the encoding operations for the toy code, which is already shown in Fig. 4. Then, the right part shows the scheduled XOR operations. The lines connecting the two parts indicate how the scheduling algorithm organizes the operations. Each XOR operation is followed by its

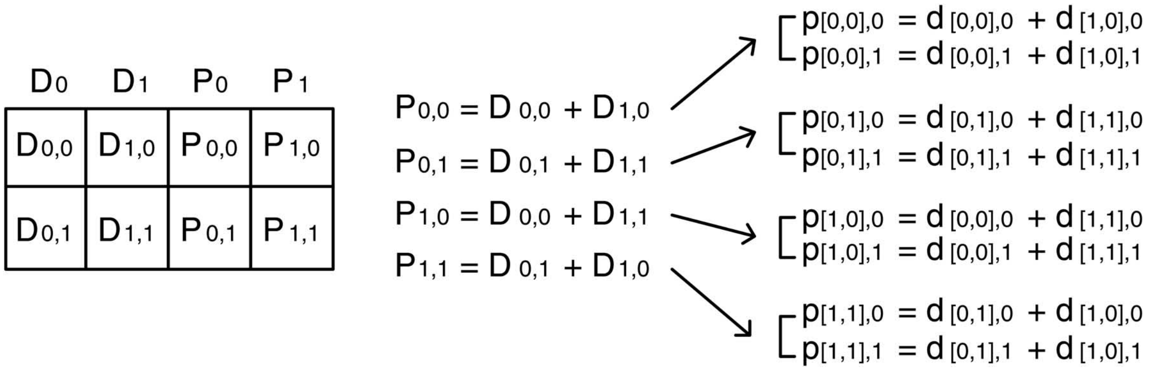
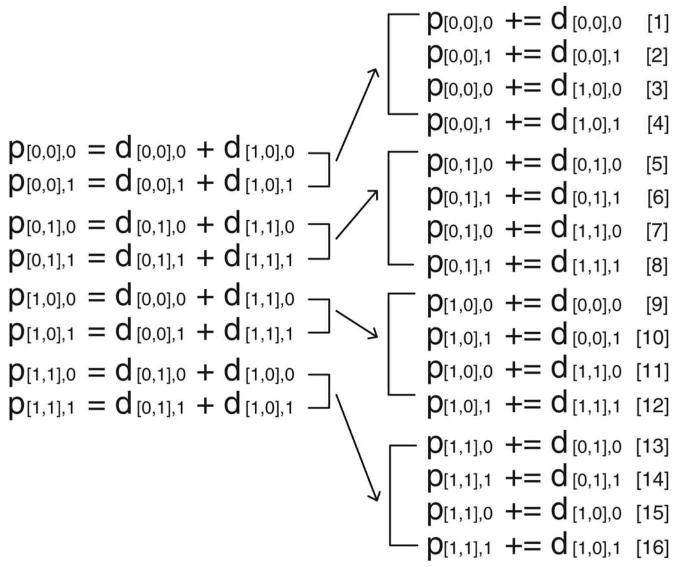

D0
D1
P0
P1
D0,0
D1,0
P0,0
P1,0
D0,1
D1,1
P0,1
P1,1
P0,0 = D0,0 + D1,0
P0,1 = D0,1 + D1,1
P1,0 = D0,0 + D1,1
P1,1 = D0,1 + D1,0
p[0,0],0 = d[0,0],0 + d[1,0],0
p[0,0],1 = d[0,0],1 + d[1,0],1
p[0,1],0 = d[0,1],0 + d[1,1],0
p[0,1],1 = d[0,1],1 + d[1,1],1
p[1,0],0 = d[0,0],0 + d[1,1],0
p[1,0],1 = d[0,0],1 + d[1,1],1
p[1,1],0 = d[0,1],0 + d[1,0],0
p[1,1],1 = d[0,1],1 + d[1,0],1

Fig. 4. A toy example code. The left part and the middle one show the code construction. The right part lists the encoding operations with the packet size of 2, i.e., each packet contains two machine words.



$p_{[0,0],0} = d_{[0,0],0} + d_{[1,0],0}$
$p_{[0,0],1} = d_{[0,0],1} + d_{[1,0],1}$
$p_{[0,1],0} = d_{[0,1],0} + d_{[1,1],0}$
$p_{[0,1],1} = d_{[0,1],1} + d_{[1,1],1}$
$p_{[1,0],0} = d_{[0,0],0} + d_{[1,1],0}$
$p_{[1,0],1} = d_{[0,0],1} + d_{[1,1],1}$
$p_{[1,1],0} = d_{[0,1],0} + d_{[1,0],0}$
$p_{[1,1],1} = d_{[0,1],1} + d_{[1,0],1}$
$p_{[0,0],0} \mathrel{+}= d_{[0,0],0}$ [1]
$p_{[0,0],1} \mathrel{+}= d_{[0,0],1}$ [2]
$p_{[0,0],0} \mathrel{+}= d_{[1,0],0}$ [3]
$p_{[0,0],1} \mathrel{+}= d_{[1,0],1}$ [4]
$p_{[0,1],0} \mathrel{+}= d_{[0,1],0}$ [5]
$p_{[0,1],1} \mathrel{+}= d_{[0,1],1}$ [6]
$p_{[0,1],0} \mathrel{+}= d_{[1,1],0}$ [7]
$p_{[0,1],1} \mathrel{+}= d_{[1,1],1}$ [8]
$p_{[1,0],0} \mathrel{+}= d_{[0,0],0}$ [9]
$p_{[1,0],1} \mathrel{+}= d_{[0,0],1}$ [10]
$p_{[1,0],0} \mathrel{+}= d_{[1,1],0}$ [11]
$p_{[1,0],1} \mathrel{+}= d_{[1,1],1}$ [12]
$p_{[1,1],0} \mathrel{+}= d_{[0,1],0}$ [13]
$p_{[1,1],1} \mathrel{+}= d_{[0,1],1}$ [14]
$p_{[1,1],0} \mathrel{+}= d_{[1,0],0}$ [15]
$p_{[1,1],1} \mathrel{+}= d_{[1,0],1}$ [16]

Fig. 5. The conventional algorithm on the toy example.

ID. In this case, the ID number indeed shows the execution order. Throughout our examples, the same XOR operation may appear in different positions in different scheduling algorithms, but its ID will remain constant.

The conventional algorithm can be summarized by two following characteristics.

1. XOR operations are guided by the order of *parity* packets.
2. The encoding iteration is performed at the *packet* level.

The first characteristic indicates that this algorithm computes parity packets one by one, to completion. This is a straightforward implementation. This characteristic means that only when one parity packet is finished computing, the next one will be calculated. For example, in Fig. 5, all the computation of $P_{[0,0]}$ appear ahead of $P_{[0,1]}$. The second characteristic is that when calculating a parity packet, each related data packet is accessed in its entirety before the next data packet is accessed. For example, during the calculation of of parity packet $P_{[0,0]}$, all words of parity packet $P_{[0,0]}$ are calculated against data packet $D_{[0,0]}$, and then they are calculated again against $D_{[0,1]}$. This is to achieve good spatial locality for both data packets and parity packets. Due to the two characteristics, this algorithm is termed Parity Packet Guided (*PPG*) XOR-scheduling. A pseudocode description of *PPG* XOR-scheduling is shown in Algorithm 1.

Algorithm 1 Parity Packet Guided (PPG) XOR-scheduling

```
INPUT: Data columns $D$, each one is a byte array
OUTPUT: Parity columns $P$, each one is a byte array
1: for each parity column $P_j$ do
2:     for each parity packet $P_{j,i}$ in column $P_j$ do
3:         for each data packet $D_{u,v}$ needed by $P_{j,i}$ do
4:             for $\mathrm{t} = 0; \mathrm{t} < \text{packet size}; \mathrm{t}{+}{+}$ do
5:                 $p_{[j,i],t} \mathrel{+}= d_{[u,v],t}$;
6:             end for
7:         end for
8:     end for
9: end for
```

Cache behavior analysis: *PPG* XOR-scheduling has good spatial locality because it sequentially accesses the words in a packet as well as the packets in one column.

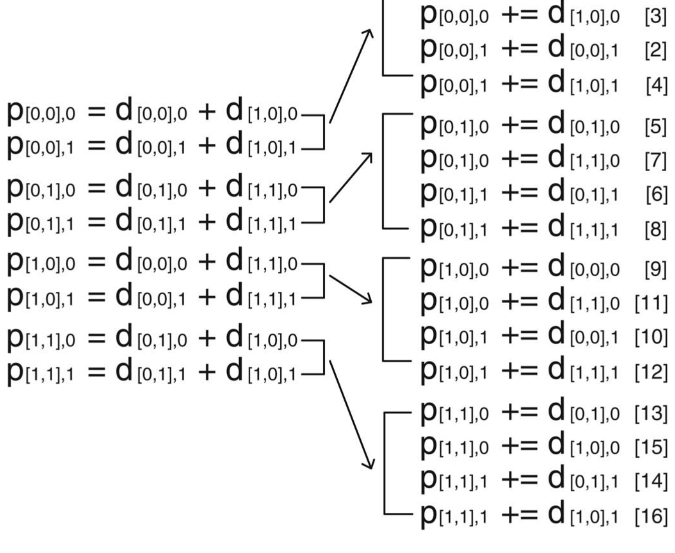
4.3 Parity Words Guided (PWG) XOR-Scheduling

A variant of *PPG* XOR-scheduling is Parity Words Guided (*PWG*) XOR-scheduling. The same as *PPG* XOR-scheduling, *PWG* XOR-scheduling still keeps computing parity packets one by one. However, the difference is during the computation of one parity packet, *PWG* XOR-scheduling calculates each parity word of a packet in its entirety before moving onto the next word. This is illustrated in Fig. 6, generated by *PWG* XOR-scheduling.

In Fig. 6, the overall computation order of parity packets is $\{P_{[0,0]},\ P_{[0,1]},\ P_{[1,0]},\ P_{[1,1]}\}$. This is also the exact order in *PPG* XOR-scheduling. The difference is in the computation of each single parity packet. For example, the first four equations in Fig. 6 show the computation sequence for parity packet $P_{[0,0]}$ is: $\{p_{[0,0],0},\ p_{[0,0],0},\ p_{[0,0],1},\ p_{[0,0],1}\}$, and then all the computation of $p_{[0,0],0}$ appears ahead of $p_{[0,0],1}$. But in Fig. 5, for the same $P_{[0,0]}$, the first four equations shows a different sequence: $\{p_{[0,0],0},\ p_{[0,0],1},\ p_{[0,0],0},\ p_{[0,0],1}\}$. The similar difference can be found for other parity packets. In summary, the encoding iteration of *PWG* XOR-scheduling is at *word* level, while in *PPG* XOR-scheduling, the iteration is at *packet* level. The characteristics of *PWG* XOR-scheduling can be listed as below.

1. XOR operations are guided by the order of *parity* packets.
2. The encoding iteration is performed at the *word* level.

A pseudocode description of this algorithm is shown in Algorithm 2.



$p_{[0,0],0} = d_{[0,0],0} + d_{[1,0],0}$
$p_{[0,0],1} = d_{[0,0],1} + d_{[1,0],1}$
$p_{[0,1],0} = d_{[0,1],0} + d_{[1,1],0}$
$p_{[0,1],1} = d_{[0,1],1} + d_{[1,1],1}$
$p_{[1,0],0} = d_{[0,0],0} + d_{[1,1],0}$
$p_{[1,0],1} = d_{[0,0],1} + d_{[1,1],1}$
$p_{[1,1],0} = d_{[0,1],0} + d_{[1,0],0}$
$p_{[1,1],1} = d_{[0,1],1} + d_{[1,0],1}$
$p_{[0,0],0} \mathrel{+}= d_{[0,0],0}$ [1]
$p_{[0,0],0} \mathrel{+}= d_{[1,0],0}$ [3]
$p_{[0,0],1} \mathrel{+}= d_{[0,0],1}$ [2]
$p_{[0,0],1} \mathrel{+}= d_{[1,0],1}$ [4]
$p_{[0,1],0} \mathrel{+}= d_{[0,1],0}$ [5]
$p_{[0,1],0} \mathrel{+}= d_{[1,1],0}$ [7]
$p_{[0,1],1} \mathrel{+}= d_{[0,1],1}$ [6]
$p_{[0,1],1} \mathrel{+}= d_{[1,1],1}$ [8]
$p_{[1,0],0} \mathrel{+}= d_{[0,0],0}$ [9]
$p_{[1,0],0} \mathrel{+}= d_{[1,1],0}$ [11]
$p_{[1,0],1} \mathrel{+}= d_{[0,0],1}$ [10]
$p_{[1,0],1} \mathrel{+}= d_{[1,1],1}$ [12]
$p_{[1,1],0} \mathrel{+}= d_{[0,1],0}$ [13]
$p_{[1,1],0} \mathrel{+}= d_{[1,0],0}$ [15]
$p_{[1,1],1} \mathrel{+}= d_{[0,1],1}$ [14]
$p_{[1,1],1} \mathrel{+}= d_{[1,0],1}$ [16]

Fig. 6. Parity Words Guided XOR-scheduling on the toy example.

Algorithm 2 Parity Words Guided (PWG) XOR-scheduling

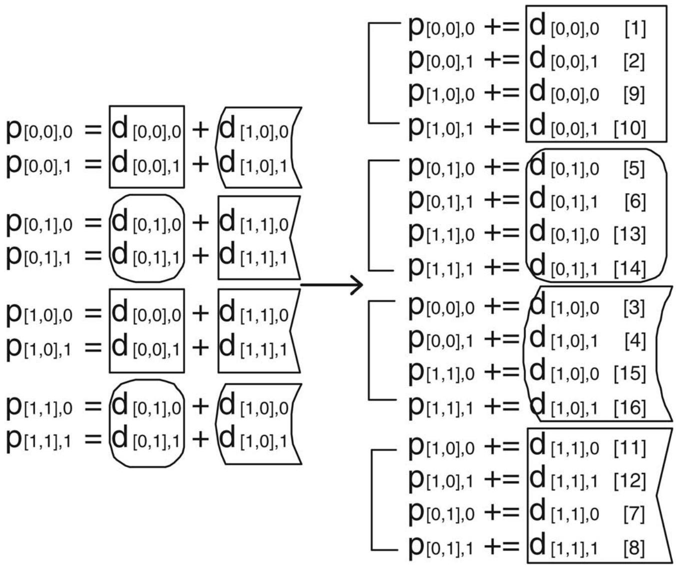
```
INPUT: Data columns $D$, each one is a byte array
OUTPUT: Parity columns $P$, each one is a byte array
1: for each parity column $P_j$ do
2:     for each parity packet $P_{j,i}$ in column $P_j$ do
3:         for $t = 0; t < \text{packet size}; t++$ do
4:             for each data packet $D_{u,v}$ needed by $P_{j,i}$ do
5:                 $p_{[j,i],t} += d_{[u,v],t}$;
6:             end for
7:         end for
8:     end for
9: end for
```

Cache behavior analysis: *PWG* XOR-scheduling reverses the innermost two loops of *PPG* XOR-scheduling. On one hand, this loop transformation improves the temporal locality of $p_{[j,i],t}$ because now $p_{[j,i],t}$ is accessed repeatedly in the innermost iteration (in line 5). On the other hand, it may lose the spatial locality of $d_{[u,v],t}$ because it accesses $d_{[u,v],t}$ in a less sequential way than *PPG* XOR-scheduling. The overall data locality will be measured experimentally in Section 5.

4.4 Data Packets Guided (DPG) XOR-Scheduling

The above two XOR scheduling algorithms (*PPG* and *PWG* XOR-scheduling) follow the intuitive idea that parity packets should be produced one by one. However, the encoding process can be reordered from another perspective, i.e., how *data* packets are consumed. This approach will yield completely different scheduling results. An algorithm named Data Packets Guided (*DPG*) XOR-scheduling is based on this idea.

The scheduling results generated by (*DPG*) XOR-scheduling is shown in Fig. 7. Now, the relationship between the equations and the scheduled operations is not straightforward. To make it easy for explanation, all data words in one data packet are grouped in one shape. Because the code contains totally four data packets, there are four different shapes. The shapes presented on the two sides of the figure actually indicate the mapping from the equations to the scheduled results. It can be seen that the XOR operations are not simply expanded from the equations; rather they follow a scheduling pattern that is significantly different from those in Figs. 5 and 6. First, the overall scheduling is guided by the order of how data packets participate in the computation. In this example, the order is $\{D_{[0,0]}, \quad D_{[0,1]}, \quad D_{[1,0]}, \quad D_{[1,1]}\}$, and each represents one shape. Then, only after all contents of one data packet is consumed, the next one will be accessed. Second, during the process of one data packet, its contents are first entirely iterated to compute for one parity packet, and then they are iterated again for the next parity packet. In the case of $D_{[0,0]}$, it is first accessed for $P_{[0,0]}$, and then accessed for $P_{[1,0]}$.



$p_{[0,0],0} = d_{[0,0],0} + d_{[1,0],0}$
$p_{[0,0],1} = d_{[0,0],1} + d_{[1,0],1}$
$p_{[0,1],0} = d_{[0,1],0} + d_{[1,1],0}$
$p_{[0,1],1} = d_{[0,1],1} + d_{[1,1],1}$
$p_{[1,0],0} = d_{[0,0],0} + d_{[1,1],0}$
$p_{[1,0],1} = d_{[0,0],1} + d_{[1,1],1}$
$p_{[1,1],0} = d_{[0,1],0} + d_{[1,0],0}$
$p_{[1,1],1} = d_{[0,1],1} + d_{[1,0],1}$
$p_{[0,0],0} += d_{[0,0],0}$ [1]
$p_{[0,0],1} += d_{[0,0],1}$ [2]
$p_{[1,0],0} += d_{[0,0],0}$ [9]
$p_{[1,0],1} += d_{[0,0],1}$ [10]
$p_{[0,1],0} += d_{[0,1],0}$ [5]
$p_{[0,1],1} += d_{[0,1],1}$ [6]
$p_{[1,1],0} += d_{[0,1],0}$ [13]
$p_{[1,1],1} += d_{[0,1],1}$ [14]
$p_{[0,0],0} += d_{[1,0],0}$ [3]
$p_{[0,0],1} += d_{[1,0],1}$ [4]
$p_{[1,1],0} += d_{[1,0],0}$ [15]
$p_{[1,1],1} += d_{[1,0],1}$ [16]
$p_{[1,0],0} += d_{[1,1],0}$ [11]
$p_{[1,0],1} += d_{[1,1],1}$ [12]
$p_{[0,1],0} += d_{[1,1],0}$ [7]
$p_{[0,1],1} += d_{[1,1],1}$ [8]

Fig. 7. Data Packets Guided XOR-scheduling on the toy example.

The (*DPG*) XOR-scheduling characteristics can be summarized as follows. Then, a pseudocode description of this algorithm is shown in Algorithm 3.

1. XOR operations are guided by the order of *data* packets.
2. The encoding iteration is performed at data *packet* level.

Cache behavior analysis: The purpose of *DPG* XOR-scheduling is to improve the locality of data packets. Now, in *DPG* XOR-scheduling, the outermost loop is on data columns, and the access to data packets is in a more sequential way than that of *PPG* XOR-scheduling. In a typical RAID-6 system, $m = 2$ and $k \geq 10$, so m is much smaller than k. Thus, the locality of data packets is more important than that of parity packets. Hence, this transformation is likely to improve the overall data locality of *DPG*.

4.5 Data Words Guided (DWG) XOR-Scheduling

Similar to the construction of *PWG* XOR-scheduling from *PPG* XOR-scheduling, a variant of *DPG* XOR-scheduling is developed. We call it Data Words Guided (*DWG*) XOR-scheduling. The same as *DPG* XOR-scheduling, *DWG* XOR-scheduling also follows the order of how data packets are consumed. The difference is how the data words in one data packet are accessed.

Algorithm 3 Data Packets Guided (DPG) XOR-scheduling

```
INPUT: Data columns $D$, each one is a byte array
OUTPUT: Parity columns $P$, each one is a byte array
1: for each data column $D_j$ do
2:     for each data packet $D_{j,i}$ in column $D_j$ do
3:         for each parity packet $P_{u,v}$ that needs $D_{j,i}$ do
4:             for $t = 0; t < \text{packet size}; t++$ do
5:                 $p_{[u,v],t} += d_{j,i],t}$;
6:             end for
```

```
7:         end for
8:     end for
9: end for
```

Let's observer an example shown in Fig. 8. This example illustrates the result of *DWG* XOR-scheduling on the toy example. In Fig. 8, the order of accessing data packets is still $\{D_{[0,0]}, \quad D_{[0,1]}, \quad D_{[1,0]}, \quad D_{[1,1]}\}$. There is no difference from *DPG* XOR-scheduling. However, during the consumption of one data packet, each data word is used for all parity calculations before moving onto the next data word. For example, in the case of $D_{[0,0]}$, $d_{[0,0],0}$ is used to calculate $p_{[0,0],0}$ and $p_{[1,0],0}$ before $d_{[0,0],1}$ is touched. Then, all the access of $d_{[0,0],0}$ appears ahead of $d_{[0,0],1}$. The algorithm's characteristics are listed below:

1. XOR operations are guided by the order of *data* packets.
2. The encoding iteration is performed at data *word* level.

A pseudocode description of this algorithm is shown in Algorithm 4.

Cache behavior analysis: *DWG* XOR-scheduling differs with *DPG* XOR-scheduling in the conversion of the innermost two loops. This transformation is to improve temporal locality of $d_{[j,i],t}$. Note that *DWG* XOR-scheduling may reduce some spatial locality of $p_{[u,v],t}$ and $d_{[j,i],t}$, but our observation is that the gain of temporal locality of $d_{[j,i],t}$ brought by the transformation is usually greater than the loss of spatial locality, and thus the overall data locality is better. Again, this is because in most storage systems, m is a small value, such as $m = 2$ in RAID-6 systems, and most data and parity words will remain in the cache across iterations of the innermost loop (in line 4).

Algorithm 4 Data Words Guided (DWG) XOR-scheduling

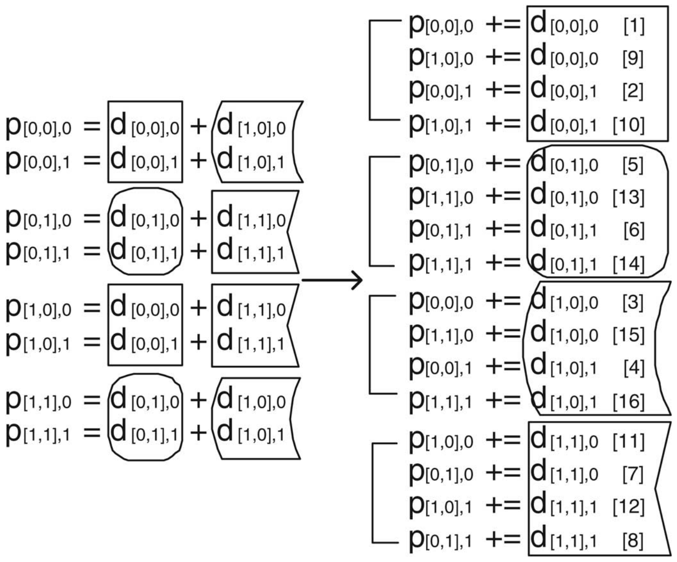
```
INPUT: Data columns $D$, each one is a byte array
OUTPUT: Parity columns $P$, each one is a byte array
1: for each data column $D_j$ do
2:     for each data packet $D_{j,i}$ in column $D_j$ do
3:         for $t = 0; t < \text{packet size}; t++$ do
4:             for each parity packet $P_{u,v}$ that needs $D_{j,i}$ do
5:                 $p_{[u,v],t} \mathrel{+}= d_{[j,i],t}$;
6:             end for
7:         end for
8:     end for
9: end for
```

4.6 Implementation Details

Generating and using schedules that are as detailed as those in Figs. 5–8 takes too much space, but is not necessary because a list of each data packet's associated parity 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 the *DPG* and *DWG* XOR-schedulings is how to initialize parity packets. They cannot be simply initialized to zero, since that increases the number of XOR operations. Instead, we copy the first data packets that are used for each parity packet, instead of XOR-ing them. Implementationally, this is simple in the *PPG* and *PWG* XOR-scheduling, where the first data packets can be easily identified at the beginning. In the *DPG* and *DWG* XOR-scheduling, it is more complex, since for a data packet, we need to know whether it is the first data packet of one parity packet. Fortunately, all the codes addressed in this paper 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, which must be calculated as an intermediate sum for every packet in parity column P_1. To handle this, all the XOR-scheduling algorithms perform two passes. In the first pass, the data packets are used to calculate P_0, P_1 and S, and then a second pass XOR's S with the packets of P_1. The STAR code, an extension of EVENODD, is handled similarly. In RDP, all but one of the packets in P_1 are calculated using packets in P_0 in addition to data packets. For that reason, two passes are also performed in RDP: a first one that calculates P_0 and P_1 without the P_0 packets, and a second one that XOR's the P_0 packets into P_1.



$p_{[0,0],0} = d_{[0,0],0} + d_{[1,0],0}$
$p_{[0,0],1} = d_{[0,0],1} + d_{[1,0],1}$
$p_{[0,1],0} = d_{[0,1],0} + d_{[1,1],0}$
$p_{[0,1],1} = d_{[0,1],1} + d_{[1,1],1}$
$p_{[1,0],0} = d_{[0,0],0} + d_{[1,1],0}$
$p_{[1,0],1} = d_{[0,0],1} + d_{[1,1],1}$
$p_{[1,1],0} = d_{[0,1],0} + d_{[1,0],0}$
$p_{[1,1],1} = d_{[0,1],1} + d_{[1,0],1}$
$p_{[0,0],0} \mathrel{+}= d_{[0,0],0}$ [1]
$p_{[1,0],0} \mathrel{+}= d_{[0,0],0}$ [9]
$p_{[0,0],1} \mathrel{+}= d_{[0,0],1}$ [2]
$p_{[1,0],1} \mathrel{+}= d_{[0,0],1}$ [10]
$p_{[0,1],0} \mathrel{+}= d_{[0,1],0}$ [5]
$p_{[1,1],0} \mathrel{+}= d_{[0,1],0}$ [13]
$p_{[0,1],1} \mathrel{+}= d_{[0,1],1}$ [6]
$p_{[1,1],1} \mathrel{+}= d_{[0,1],1}$ [14]
$p_{[0,0],0} \mathrel{+}= d_{[1,0],0}$ [3]
$p_{[1,1],0} \mathrel{+}= d_{[1,0],0}$ [15]
$p_{[0,0],1} \mathrel{+}= d_{[1,0],1}$ [4]
$p_{[1,1],1} \mathrel{+}= d_{[1,0],1}$ [16]
$p_{[1,0],0} \mathrel{+}= d_{[1,1],0}$ [11]
$p_{[0,1],0} \mathrel{+}= d_{[1,1],0}$ [7]
$p_{[1,0],1} \mathrel{+}= d_{[1,1],1}$ [12]
$p_{[0,1],1} \mathrel{+}= d_{[1,1],1}$ [8]

Fig. 8. Data Words Guided XOR-scheduling on the toy example.

4.7 Compiler Optimization

Compiler optimization is an important technique to improve an algorithm's performance during the compilation. However, even with the optimization, the performance of the four XOR-scheduling algorithms would be still greatly different due to the following reasons:

1. Each algorithm contains four nested loops. Such a complicated loop structure makes it hard for a compiler to generate similar scheduling operations, if possible.
2. For easy explanation, the presented pseudocode is a simplified version of its source code. In fact, a real implementation is filled with many details. The details may obscure the loop structure and complicate the XOR-scheduling results generation process.

TABLE 1
Test Platform Configurations

Platform	CPU Model	Speed	L1 Cache	L2 Cache	Memory * Channels	gcc
P4	Pentium 4	3.0GHz	28KB	2MB	DDR 533MHz * 1	4.4.1
Pd	Pentium Dual Core	2.8GHz	$2 \cdot 16$KB	$2 \cdot 1$MB	DDR2 533MHz * 1	4.2.1
Pc2d	Pentium Core 2 Duo	2.1GHz	$2 \cdot 32$KB	3MB	DDR2 667MHz * 1	4.1.3
Pc2q	Pentium Core 2 Quad	2.4GHz	$4 \cdot 32$KB	$2 \cdot 4$MB	DDR2 800MHz * 1	4.3.2

3. Although Algorithm 3 and 4 look like Algorithm 1 and 2, they are significantly different: Algorithm 3 and 4 identify the related parity packets for a data packet; however, Algorithm 1 and 2 keep track of the related data packets for a parity packet. Obviously, they are two very different functions. It is difficult for a compiler to derive one from another.

Our observation is also consistent with the results from Lebeck et al. [26], who found that manual optimization is still needed to improve an algorithm's performance.

5 Performance Evaluation

Section 4 introduces the four XOR-scheduling algorithms and demonstrates how they work on a toy example code. To evaluate their performance, we have implemented the algorithms for real erasure codes and measured the performance on a variety of platforms for a comprehensive view of the performance. The erasure codes include: the Liberation codes [20], EVENODD [21], RDP [8], and the STAR code [31]. All are RAID-6 codes ($m = 2$), with the exception of the STAR code which tolerates three failures ($m = 3$).

The XOR-scheduling algorithms' performance is measured through two sets of experiments. The first set is presented in Sections 5.2 and 5.4. This set is from a system practitioner's point of view. We treat CPU cache as a black box and measure the encoding performance. The second set is presented in Section 5.3. This set is to further examine the interactions of the scheduling algorithms with the memory hierarchy of test platforms. This verifies that the encoding performance gains of the scheduling algorithms do indeed come from their effective use of CPU caches. Instead of simulations or simplistic modeling, we choose to perform profiling CPU events to truly reflect real cache behavior.

5.1 Experiment Setup

Since CPU cache behavior is complicated, we have run experiments on various platforms to better understand the previously described scheduling algorithms. The experiments are conducted on four different platforms, whose detailed configurations are shown in Table 1.

All platforms run 64-bit processors, and they are installed with 64-bit version of Linux. Since 64-bit platforms perform XOR operations on 64-bit words, their encoding performance is as much as a factor of two faster than their 32-bit counterparts [15]. The implementations of all algorithms do not use complicated performance enhancement techniques, such as SIMD (single instruction multiple data) instructions [32] or software prefetching [33]. These techniques may speed up XOR operations or improve CPU cache behavior, and they are currently being explored by other researchers.

Our source code is written in C and compiled using **gcc**. We tried both **-O2** and **-O3** optimization flags, of which **-O2** is recommended for most applications while **-O3** is the highest level of optimization [34]. No other optimization flags are set, such as prefetch-loop-arrays [35]. Since the performance results of them are similar, we only present the results for **-O2** here. Each platform's **gcc** version is listed in the last column of Table 1. As single-threaded program is easy to implement and optimize, 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 [18] as the implementation for *PPG* XOR-scheduling, since that is exactly what Jerasure implements. For the other codes and algorithms, we used Jerasure as our base and crafted custom code from it.

Our experimental framework is very similar to that in [15]. All tests are performed in main memory, without actual disk I/O, because that introduces a great deal of variability. We encode totally a gigabyte of data, which is randomly generated. We evaluate encoding performance using the metric of *encoding throughput*, calculated with the equation below:

$$\text{Encoding throughput} = \frac{\text{Total user data size}}{\text{Encoding time}}.$$

Encoding throughput represents how quickly one can turn user data into parity data with a given code, algorithm and platform. The encoding time is calculated by the **gettimeofday()** system call. Each data point is the average of 30 test runs. The data plotted is all within a confidence interval of 95%.

5.2 Experiments on the Liberation Codes

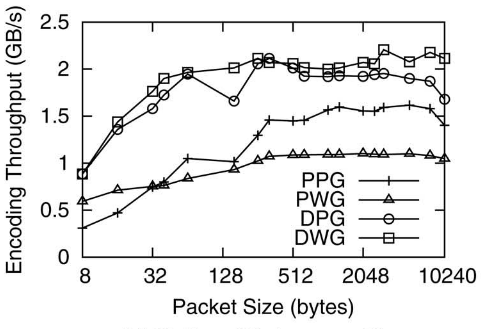
This section focuses on a performance comparison of the Liberation codes, because it is the one code for which an open source implementation is available.

5.2.1 Parameters $k = w = 11, m = 2$:

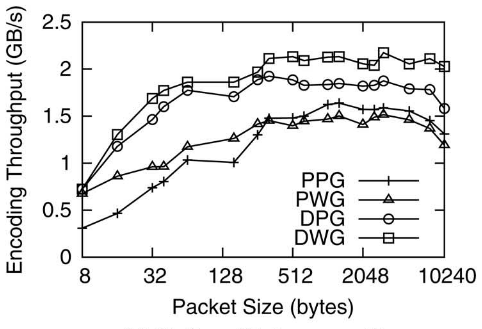
The parameters of the first experiment for the Liberation code are $k = w = 11$, and $m = 2$, as these represent a typical RAID-6 system. We vary the packet size because it can impact performance greatly [15].

Fig. 9(a) compares the algorithms' performance on platform P4. The results can be summarized as follows:

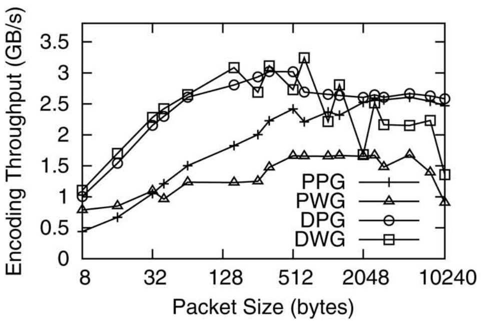
1. As packet sizes increase from a small value, the performance of all scheduling algorithms improves significantly. This mirrors the observations by Plank et al. [15].
2. For all packet sizes, *DWG* and *DPG* XOR-scheduling achieve much better encoding performance than the other two algorithms. It is because *DWG* and *DPG* XOR-scheduling have better data locality than the other two, matching the analysis given in Section 4.



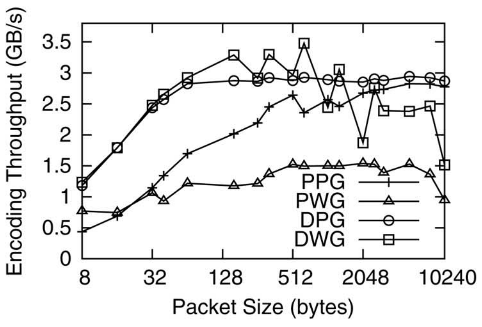
(a) Platform P4, $k = w = 11$



(b) Platform Pd, $k = w = 11$



(c) Platform Pc2d, $k = w = 11$



(d) Platform Pc2q, $k = w = 11$

Fig. 9. Encoding performance of Liberation codes, $k = w = 11$ and $m = 2$.

3. *DWG* XOR-scheduling has the highest peak performance, and *DPG* XOR-scheduling performs second best. *PPG* XOR-scheduling is the next one, and *PWG* XOR-scheduling is the last one.
4. *DWG* XOR-scheduling and *DPG* XOR-scheduling achieve their peak encoding performance with a relatively small packet size. This is useful because it provides real systems more flexibility on choosing packet sizes for high performance.

Fig. 9(b) displays the results on the Pd platform. The performance is very similar to P4, so the four above observations on P4 also apply to Pd.

Fig. 9(c) shows the results for platform Pc2d. This platform shows a new performance feature—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 *DPG* and *PPG* XOR-scheduling. This will be analyzed further in Section 5.3. Nonetheless, the two most important observations still hold: 1) the peak performance of *DWG* XOR-scheduling is higher than all others; 2) the peak performance is obtained with small packet size. Compared to *DWG* XOR-scheduling, although *DPG* XOR-scheduling has lower peak performance, it achieves more stable performance and its performance is always higher than that of *PPG* and *PWG* XOR-scheduling.

Fig. 9(d) shows the results on platform Pc2q. These are very similar to Pc2d, and the observations for Pc2d are also valid for Pc2q.

The peak performance of *DWG* and *PPG* XOR-scheduling (the conventional one) on the four platforms are summarized in Table 2. It clearly displays that *DWG* XOR-scheduling achieves significant performance improvement over *PPG* XOR-scheduling, from 23% to 36%.

5.2.2 *Modifying k and w*

In the Liberation codes, the number of parity 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. We also observed similar results for $k = w = 5$ and $k = w = 17$. Due to the page limitation, the results for them are not included here. In summary, the observations that held for $k = 11$ also hold for the smaller and larger values of k. Thus, all scheduling algorithms are stable for this code among this range of parameters.

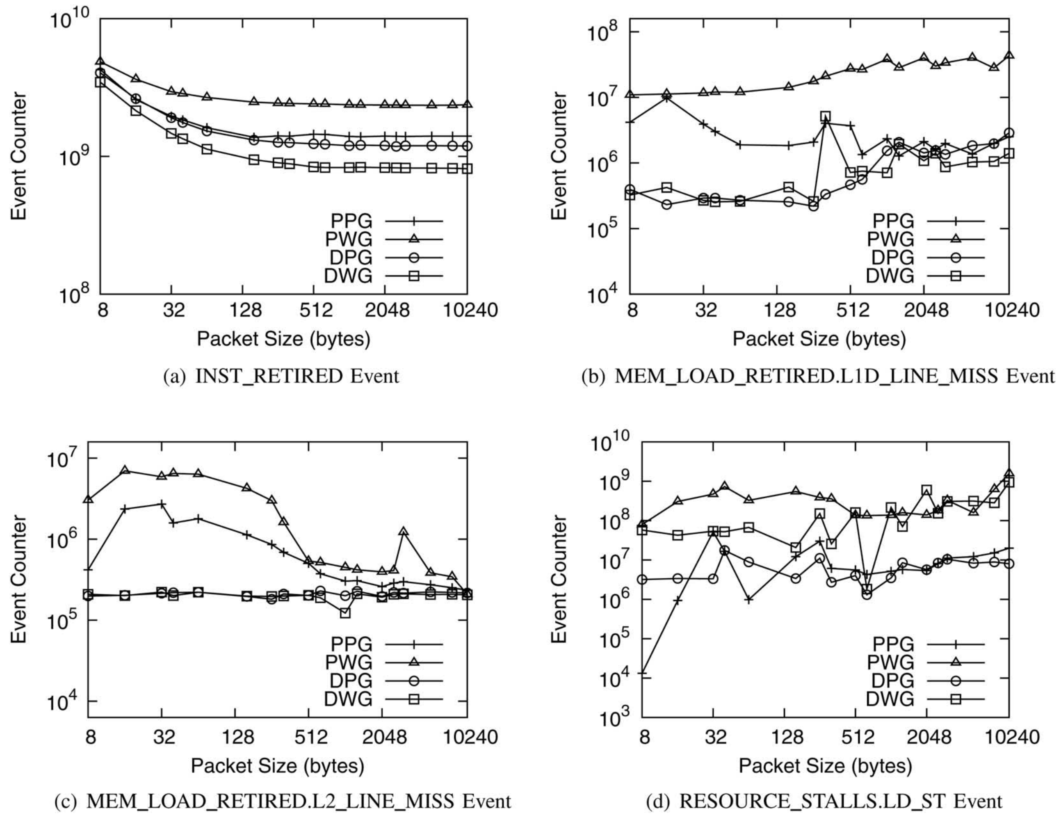
It is worth noting that the values of k are prime numbers in above experiments. In practice, this is not required. A well known technique called shortening [22], [16] can be applied to produce a shorter code but with a certain efficiency loss in space utilization.

5.3 Performance Profiling for the Liberation Code

To better understand the interactions of XOR-scheduling algorithms and the memory hierarchy of test platforms, we performed a profiling experiment with the Liberation Codes with $k = w = 11$. We used the Intel VTune performance

TABLE 2
Comparison of Peak Encoding Performance of Liberation Codes for $k = w = 11$ and $m = 2$

Platform	DWG	PPG	Improvement
P4	2.207 GB/s	1.616 GB/s	36%
Pd	2.174	1.64	32%
Pc2d	3.243	2.607	24%
Pc2q	3.479	2.824	23%



(a) INST_RETIRED Event (b) MEM_LOAD_RETIRED.L1D_LINE_MISS Event (c) MEM_LOAD_RETIRED.L2_LINE_MISS Event (d) RESOURCE_STALLS.LD_ST Event

Fig. 10. Performance profiling results of Liberation codes for $k = w = 11$ and $m = 2$ on platform Pc2q.

Analyzer [19], which leverages performance counting registers available on certain Intel processors. The event-based sampling (EBS) profiler of VTune allows one to register certain events for monitoring, and then VTune samples the number of these events during live runs of programs on the processor. For each event, one specifies a *sample-after-value (SAV)*. VTune maintains a hardware counter of the event, and when the counter reaches the SAV value, VTune increments a user-specified value and resets the counter. Thus, one may trade off the precision of sampling with the invasiveness of VTune.

This profiling approach differs from those based on simulation, such as Cachegrind [36]. Its main advantage is that it profiles real executions on the processors, and thus is not limited by assumptions made by a simulator. Its main disadvantage is the tradeoff between sampling granularity and profiling invasiveness. A second disadvantage is the fact that while one can count certain events, it may be hard to interpret why those events are happening. Regardless, the various event counts can give us insight into how a program is behaving.

For the XOR-scheduling algorithms, we chose to monitor the events which we summarize below. One measures instruction counts, while the remaining three assess different parts of the memory hierarchy. For complete descriptions, please see the Intel software developer's manual [37].

- **INST_RETIRED**: This counts the number of instructions that are retired. On processors with speculative execution, this is a measure of the instructions that have completed successfully and had their results written to the cache.
- **MEM_LOAD_RETIRED.L1D_LINE_MISS**: This is a measure of how many load operations miss the L1 data cache and send a request to the L2 cache to satisfy the load operation.
- **MEM_LOAD_RETIRED.L2_LINE_MISS**: This is a measure of how many L2 cache requests are unsatisfied and must be reloaded from main memory.
- **RESOURCE_STALLS.LD_ST**: This is an event when the processor stalls because the load and store buffers used for pipelined and out-of-order execution become full.

For the INST_RETIRED event, we set the SAV value to 100,000. For all others, we set it to 10,000.

We show the results of the profiling tests in Fig. 10. The platforms Pc2d and Pc2q support the profiling of the above events. Their profiling results are similar, so we only display Pc2q in Fig. 10. It is worth noting that one cannot derive actual performance from the various event counts. Instead, the profiling helps us to gain insight into why the performance of the various algorithms is as it is. We evaluate each of the events below.

INST_RETIRED [Fig. 10(a)]: For all algorithms, as the packet size increases, the instruction counts decrease, becoming roughly constant for packet sizes greater than 512 bytes. This is because fewer loop iterations are required as the packet size increases. One thing worth noting is that their event counts should be similar across the scheduling algorithms because the algorithms should be only different in their cache behavior. However, their counts are not very close. Of the four algorithms, the *PWG* algorithm retires the most instructions and *DWG* the least. The other two exhibit similar performance. One possible reason could be there is correlation between the number of instructions and the cache efficiency. We will explore it in our future work.

TABLE 3
Encoding Parameters of Various Codes

Code	k	w	m
EVENODD	11	10	2
RDP	10	10	2
STAR	11	10	3

MEM_LOAD_RETIRED.L1D_LINE_MISS [Fig. 10(b)]: This measures the number of processor stalls due to L1 cache misses. As the packet sizes increase, these values increase too, because fewer packets fit into the L1 cache. The *PWG* algorithm displays significantly more stalls due to L1 cache misses than the others, which exhibit similar numbers as the packet sizes grow to 1 K and beyond.

MEM_LOAD_RETIRED.L2_LINE_MISS [Fig. 10(c)]: This measures the number of processor stalls due to L2 cache misses. Since these misses must be satisfied from main memory, the impact of these stalls is greater than for L1 misses. The *DWG* and *DPG* algorithms exhibit stable performance with respect to L2 cache misses, and this performance is independent of the packet size. Moreover, they are better than the *PPG* and *PWG* algorithms for all packet sizes. This is the main contributing factor to the *DWG* and *DPG* algorithms' superior performance in Fig. 9(d). The *PPG* and *PWG* algorithms have fewer L2 cache misses as their packet sizes increase.

RESOURCE_STALLS.LD_ST [Fig. 10(d)]: In this figure, the *PPG* and *DPG* algorithms exhibit the fewest stalls due to the load and store buffers being full. The *DWG* algorithm shows the greatest variability here, especially for large packet sizes. This is reflected in its variable overall performance in Fig. 9(d). As in the other figures, the *PWG* algorithm exhibits the worst overall performance.

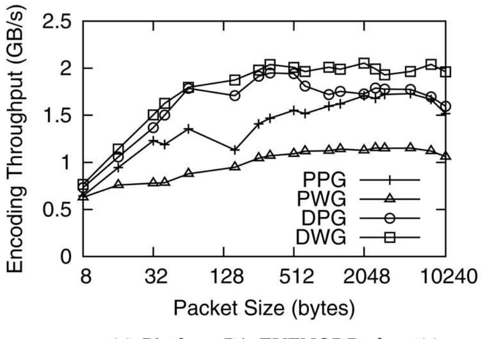
In summary, while the *DWG* algorithm achieves the best performance for each code and platform, it is more sensitive to having the packet size impact the cache behavior. This is most pronounced in Fig. 10(b) and (d). On the other hand, the *DPG* algorithm achieves a nicer blend of performance and stability. The *PWG* algorithm achieves the worst performance, as reflected in every event measured in our profiling experiments. It is worth noting that the *DPG* and *DWG* algorithms have fewer L1 and L2 cache misses at lower packet sizes, and thus both algorithms achieve their peak performance at smaller packet sizes than *PPG* and *PWG*.

It is interesting to observe that although *DPG* and *DWG* were originally designed to reduce the number of cache misses, the profiling results show that they also effectively decrease the number of executed instructions. Regardless, it is significant to note that the scheduling of XOR operations indeed affects all of these performance events.

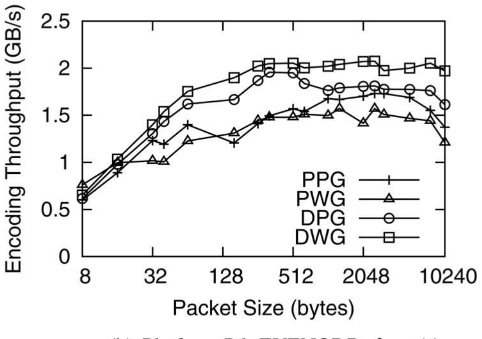
Finally, we did profile the other erasure codes that are discussed below. However, since their results were very similar to the Liberation code profiling, we omit their presentation here.

5.4 Experiments on Other Erasure Codes

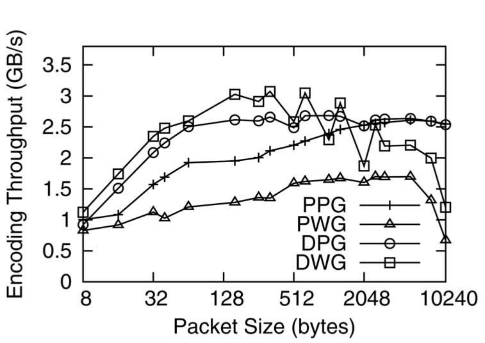
To compare the algorithms on other codes, this section tests EVENODD, RDP, and the STAR code. Again, all but the STAR code are RAID-6 codes. Among them, EVENODD and RDP have efficient decoding algorithms, while RDP has better encoding performance and EVENODD has better update performance [16]. The STAR 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 3.



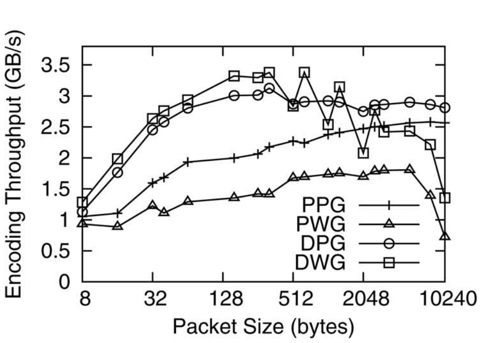
(a) Platform P4, EVENODD, $k = 11$



(b) Platform Pd, EVENODD, $k = 11$

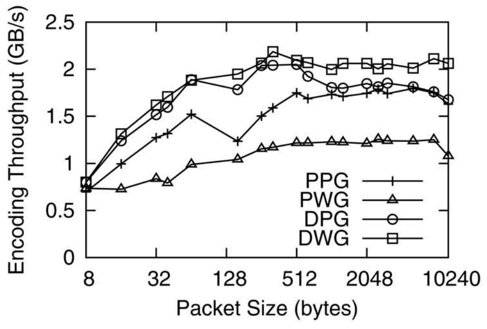


(c) Platform Pc2d, EVENODD, $k = 11$

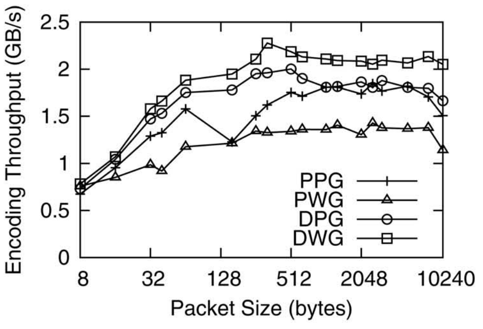


(d) Platform Pc2q, EVENODD, $k = 11$

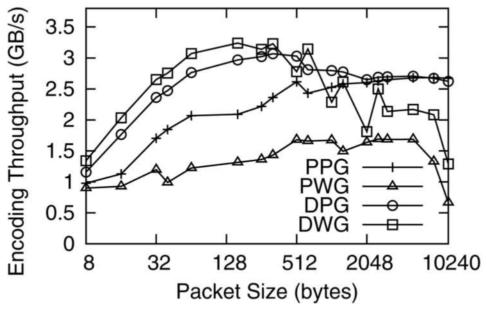
Fig. 11. Encoding performance of EVENODD for $k = 11$, $w = 10$ and $m = 2$.



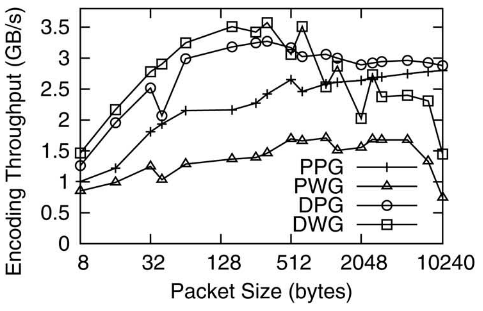
(a) Platform P4, RDP, $k = 10$



(b) Platform Pd, RDP, $k = 10$



(c) Platform Pc2d, RDP, $k = 10$

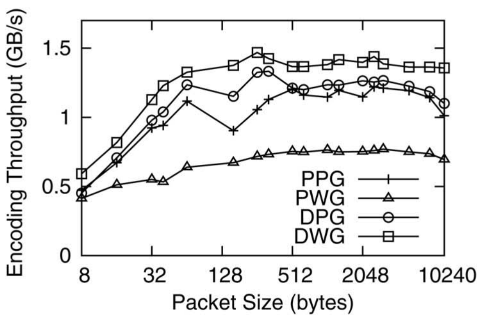


(d) Platform Pc2q, RDP, $k = 10$

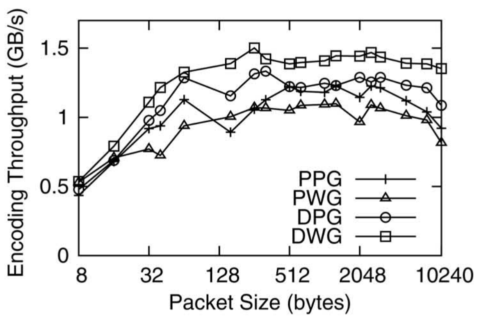
Fig. 12. Encoding performance of RDP for $k = w = 10$ and $m = 2$.

The results are in Figs. 11–13. In terms of peak performance, the codes match expectations. When $k = w$, the Liberation codes' performance is theoretically identical to EVENODD [20], and this is reflected in the results. RDP should encode the fastest, and it does. The STAR code's performance is worse than the others because it encodes an extra parity device.

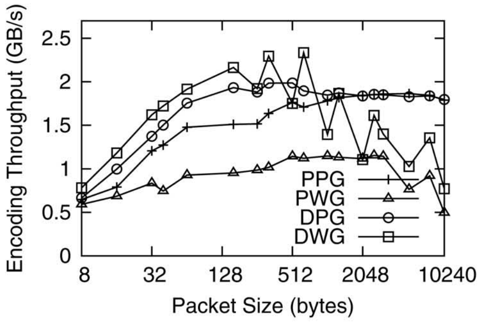
In terms of the trends, the codes' performance matches the Liberation codes, and all the observations for the Liberation codes hold for these codes as well. The results show that the



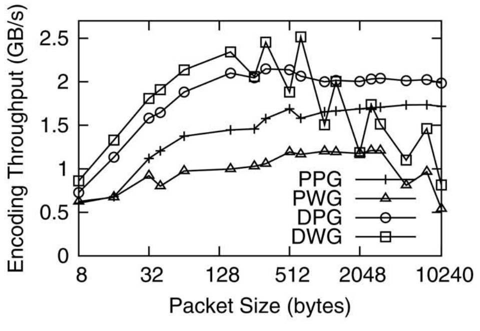
(a) Platform P4, STAR, $k = 11$



(b) Platform Pd, STAR, $k = 11$



(c) Platform Pc2d, STAR, $k = 11$



(d) Platform Pc2q, STAR, $k = 11$

Fig. 13. Encoding performance of the STAR code for $k = 11$, $w = 10$, and $m = 3$.

TABLE 4
Encoding Performance of RDP for $k = w = 10$ and $m = 2$

Platform	DWG XOR-scheduling	Reported in [8]
P4	2.186 GB/s	1.55 GB/s

performance comparison of the scheduling algorithms is consistent for a wide variety of codes.

5.5 Encoding Performance of RDP

In [8], Corbett et al. provide encoding performance of RDP code, and it is the best reported result of RDP in the literature. The platform used in [8] contains two Pentium 4 CPUs of 2.8 GHz, and one CPU is dedicated for RDP encoding. They do not report the platform word size and the cache sizes of the CPU. The most similar platform in our tests is platform P4, but P4 only has one CPU. The performance comparison of *DWG* XOR-scheduling with their reported performance is given in Table 4.

Table 4 shows that the encoding performance achieved by *DWG* XOR-scheduling is much higher than their reported performance. Without knowing their exact implementation details, the reasons can be manyfold. However, a simple conclusion to draw is that *DWG* XOR-scheduling, an algorithm proposed in this paper, can achieve much higher performance than existing reported results.

6 Conclusions

This paper studies efficient XOR-scheduling algorithms 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. Two new proposed algorithms, named Data Words Guided XOR-scheduling and Data Packets Guided XOR-scheduling, are able to efficiently utilize CPU cache and thus achieve much better encoding performance than the conventional algorithm. Through performance evaluation on some widely known erasure codes on a variety of platforms, we show that the encoding performance obtained by Data Words Guided XOR-scheduling considerably outperforms that of the conventional algorithm; although Data Packets Guided XOR-scheduling improves the performance less significantly than Data Words Guided XOR-scheduling, it has more stable performance across various packet sizes on different platforms.

In general, as the amount of data is accessed much more than that of parity in the encoding process, preserving data locality is more critical for high performance. For platforms with a more flexible cache design, such as RAID-controllers, packet size should be an important factor to consider for data prefetch size. It is desirable that two sizes are matched.

References

[1] 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 *Proc. 9th Int. Conf. Archit. Support Program. Languages Oper. Syst. (ASPLOS'00)*, Dec. 2000, pp. 190–201.

[2] N. Joukov, A. M. Krishnakumar, C. Patti, A. Rai, S. Satnur, A. Traeger, and E. Zadok, "RAIF: Redundant array of independent filesystems," in *Proc. 24th IEEE Conf. Mass Storage Syst. Technol. (MSST'07)*, Sep. 2007, pp. 199–212.

[3] 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 Trans. Parallel Distrib. Syst.*, vol. 12, no. 2, pp. 99–114, 2001.

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

[5] J. J. Wylie, and R. Swaminathan, "Determining fault tolerance of XOR-based erasure codes efficiently," in *Proc. 37th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw. (DSN'07)*, Jun. 2007, pp. 206–215.

[6] J. L. Hafner, "WEAVER codes: Highly fault tolerant erasure codes for storage systems," in *Proc. 4th USENIX Conf. File Storage Technol. (FAST'05)*, Dec. 2005, pp. 211–224.

[7] Cleversafe Inc. (2008). *Cleversafe Dispersed Storage* [Online]. Available: http://www.cleversafe.org/downloads.

[8] P. Corbett, B. English, A. Goel, T. Grcanac, S. Kleiman, J. Leong, and S. Sankar, "Row-diagonal parity for double disk failure correction," in *Proc. 3rd USENIX Conf. File Storage Technol. (FAST'04)*, Mar. 2004, pp. 1–14.

[9] L. Xu, and J. Bruck, "Low density MDS code and factors of complete graphs," *IEEE Trans. Inf. Theory*, vol. 45, pp. 1817–1826, 1999.

[10] J. Blomer, M. Kalfane, M. Karpinski, R. Karp, M. Luby, and D. Zuckerman, "An XOR-based erasure-resilient coding scheme," *Int. Comput. Sci. Inst.*, Tech. Univ. of California, Berkeley, CA, U.S., Tech. Rep. TR-95-048, Aug. 1995.

[11] I. S. Reed, and G. Solomon, "Polynomial codes over certain finite fields," *J. Soc. Ind. Appl. Math.*, vol. 8, no. 10, pp. 300–304, 1960.

[12] M. Blaum, and R. M. Roth, "On lowest density MDS codes," *IEEE Trans. Inf. Theory*, vol. 45, no. 1, pp. 46–59, Jan. 1999.

[13] D. Narayanan, E. Thereska, A. Donnelly, S. Elnikety, and A. Rowstron, "Migrating server storage to SSDS: Analysis of tradeoffs," in *Proc. 4th ACM SIGOPS/EuroSys Eur. Conf. Comput. Syst. (EuroSys'09)*, Apr. 2009, pp. 145–158.

[14] L. Xu, "X-code: MDS array codes with optimal encoding," *IEEE Trans Inf. Theory*, vol. 45, no. 1, pp. 272–276, Jan. 1999.

[15] 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 *Proc. 7th USENIX Conf. File Storage Technol. (FAST'09)*, Feb. 2009, pp. 253–265.

[16] J. S. Plank, and L. Xu, "Optimizing Cauchy reed Solomon codes for fault-tolerant network storage applications," in *Proc. 5th IEEE Int. Symp. Netw. Comput. Appl. (NCA'06)*, Jul. 2006, pp. 173–180.

[17] M. Luby. (1997). *Code for Cauchy Reed-Solomon Coding* [Online]. Available: http://www.icsi.berkeley.edu/~luby/cauchy.tar.uu.

[18] J. S. Plank, "Jerasure: A library in C/C++ facilitating erasure coding for storage applications," Univ. Tennessee, Knoxville, TN, U.S., Tech. Rep. CS-07-603, Sep. 2007.

[19] Intel. (2010). "Performance Profiler for Serial and Parallel Performance Analysis" [Online]. Available: http://software.intel.com/en-us/intel-vtune/.

[20] J. S. Plank, "The RAID-6 liberation codes," in *Proc. 6th USENIX Conf. File Storage Technol. (FAST'08)*, Feb. 2008, pp. 97–110.

[21] M. Blaum, J. Brady, J. Bruck, and J. Menon, "EVENODD: An efficient scheme for tolerating double disk failures in RAID architectures," *IEEE Trans. Comput.*, vol. 44, no. 2, pp. 192–202, Feb. 1995.

[22] F. J. MacWilliams and N. J. A. Sloane, *The Theory of Error Correcting Codes*. Amsterdam, North-Holland: North Holland Publishing Co., 1977.

[23] M. Blaum, and R. M. Roth, "New array codes for multiple phased burst correction," *IEEE Trans. Inf. Theory*, vol. 39, no. 1, pp. 66–77, Jan. 1993.

[24] C. Huang, J. Li, and M. Chen, "On optimizing XOR-based codes for fault-tolerant storage applications," in *Proc. IEEE Inf. Theory Workshop (ITW'07)*, Sep. 2007, pp. 218–223.

[25] J. L. Hafner, V. Deenadhayalan, K. K. Rao, and J. A. Tomlin, "Matrix methods for lost data reconstruction in erasure codes," in *Proc. 4th USENIX Conf. File Storage Technol. (FAST'05)*, Dec. 2005, pp. 183–196.

[26] A. R. Lebeck, and D. A. Wood, "Cache profiling and the SPEC benchmarks: A case study," *Comput.*, vol. 27, no. 10, pp. 15–26, 1994.

[27] U. Drepper. (2007). *What every programmer should know about memory. Red Hat Inc.* [Online]. Available: http://lwn.net/Articles/250967/.

[28] J. L. Hennessy, D. A. Patterson, and D. Goldberg, *Computer Architecture: A Quantitative Approach*. San Mateo, CA, USA: Morgan Kaufmann, May 2002, ch. Appendiex C.

[29] R. Allen and K. Kennedy, *Optimizing Compilers for Modern Architectures*. San Mateo, CA, USA: Morgan Kaufmann, Oct. 2001.

[30] J. Warren, "A hierarchical basis for reordering transformations," in *Proc. 11th ACM SIGACT-SIGPLAN Symp. Principles Program. Languages (POPL'84)*, Jan. 1984, pp. 272–282.

[31] C. Huang, and L. Xu, "STAR: An efficient coding scheme for correcting triple storage node failures," in *Proc. 4th USENIX Conf. File Storage Technol. (FAST'05)*, Dec. 2005, pp. 197–210.

[32] Intel. (2010). *Intel 64 and IA-32 Architectures Software Developer's Manuals, Volumn 1* [Online]. Available: http://www.intel.com/Assets/PDF/manual/253666.pdf.

[33] J. Lu, H. Chen, R. Fu, W. Hsu, B. Othmer, and P. Yew, "The performance of runtime data cache prefetching in a dynamic optimization system," in *Proc. 36th Annu. Int. Symp. Microarchit. (MICRO'03)*, Dec. 2003, pp. 180–190.

[34] gentoo wiki. (2010). CFLAGS [Online]. Available: http://en.gentoo-wiki.com/wiki/CFLAGS.

[35] T. C. Mowry, M. S. Lam, and A. Gupta, "Design and evaluation of a compiler algorithm for prefetching," in *Proc. 5th Int. Conf. Archit. Support Program. Languages Oper. Syst. (ASPLOS'92)*, Oct. 1992, pp. 62–73.

[36] Cachegrind. (2010). *Cachegrind: a cache and branch-prediction profiler* [Online]. Available: http://valgrind.org/docs/manual/cg-manual.html.

[37] Intel. (2010). *Intel 64 and IA-32 Architectures Software Developer's Manuals, Volumn 3B* [Online]. Available: http://www.intel.com/Assets/PDF/manual/253669.pdf.

Jianqiang Luo received the MS degree in computer science from Shanghai Jiao Tong University, China, in 2004, and the PhD degree in computer science from Wayne State University, Detroit, MI, in 2011. His current research interests include distributed storage systems, and error-correcting codes.

Mochan Shrestha received the BS degree in computer science from Grand Valley State University, Allendale, MI, in 2000, and the MS degree in mathematics from Michigan State University, East Lansing, in 2005. He has been a PhD student of computer science at Wayne State University, Detroit, MI, since 2006. His current research interests include storage systems, error-correcting codes, and flash memory.

Lihao Xu received the BS and MS degrees in electrical engineering from Shanghai Jiao Tong University, China, in 1988 and 1991, respectively, and the PhD degree in electrical engineering from California Institute of Technology, Pasadena, in 1999. He has been an associate professor of computer science with Wayne State University, Detroit, MI, since August 2005. His current research interests include distributed computing and storage systems, error-correcting codes, information theory and data security.

James S. Plank received the BS degree in computer science from Yale University, New Haven, CT, in 1988, and MA and PhD degrees in computer science from Princeton University, New Jersey, in 1990 and 1993, respectively. He has been a full professor with the Department of Computer Science/EECS, the University of Tennessee, Knoxville, since 2006. His current research interests include distributed systems, storage systems, erasure codes, fault-tolerance, and operating systems.