Practical Performance Study of Erasure Codes for High Performance Data Storage Systems

Rui Chen
Wayne State University
Detroit, Michigan 48202
Email: chenrui@wayne.edu

Lihao Xu
Wayne State University
Detroit, Michigan 48202
Email: lihao@wayne.edu

Abstract—As erasure codes have been widely adopted in most large scale data storage systems and applications, implementations of high performance erasure codes have been improved significantly in recent years, especially by employing Intel's Streaming SIMD Extensions (SSE) instructions. Augmenting the survey work in [24] conducted almost a decade ago, this paper compares practical performance of three open source or public domain erasure coding libraries, namely Jerasure and Intel's ISA-L for RS code, and a STAR code implementation. The goal of this paper is to provide data storage practitioners a guideline when they choose a proper erasure code for storage applications and systems that need high performance in encoding and decoding operations in the order of GBs/sec. Additionally this paper identifies a practical technique that can further improve decoding performance of RS code greatly for both Jerasure and ISA-L for the most frequent disk failure pattern, i.e., one disk failure.

I. INTRODUCTION

By now it has been well known in data storage community that erasure codes play an important role in achieving data reliability of large scale data storage systems and they are widely used in many systems, such as Amazon’s S3 [1] [20], Google’s File System [8] and its successor Colossus [6], Microsoft’s Azure [10] [14], and Facebook’s storage systems [25] [27]. Instead of direct mirroring or replication, erasure code can more economically utilize storage space and network bandwidth (when distributing data) to achieve same degree of data reliability. The cost, of course, is extra computation needed for both encoding (for data writes) and decoding (for data reads when failures occur). While it has been perceived that encoding and decoding computation speeds of fast erasure codes can achieve line speed, which is not true by our test results shown later in this paper, and that erasure code encoding and decoding operations have not become a critical bottleneck in most data storage systems, the following emerging technologies and trends indicate encoding and decoding speeds of erasure codes employed become more and more critical in affecting to deciding overall storage system performance and cost:

1) The emerging non-volatile memory (NVM) technologies are blurring speed boundary between main memroy and persistent storage, making persistent storage IO speed easily reaching GB/s [16]. It thus calls for erasure code encoding and decoding operations to at least match such IO speeds for high performance storage systems consisting of NVM arrays.

2) Most large scale data centers deploy virtual machines to satify data storage needs. As an essential component of storage system, computation savings from erasure code encoding and decoding operations can be more effectively used for other operations in the whole system, not just the storage subsystem, thus making data center more efficient and economical.

3) Further more, software defined storage systems (SDS) also call for more efficient erasure code encoding and decoding operations to improve overall system’s performance.

It has been almost a decade since last time performance of then popular open source erasure code libraries was evaluated [24]. While not many fundamentally new erasure codes have emerged since then, implementations of existing erasure codes, mainly the versatile Reed-Solomon code [26], have been improved a lot, especially by employing Intel’s Streaming SIMD Extensions (SSE) instruction set [12] for efficient finite field operations and XORs [23]. Another change in erasure code practice since then is the use of so-called Local Reconstruction Codes (LRC) [10] and Sector-Disk (SD) Codes [21] to reduce the amount of data needed to be read when recovering one disk/node failure at the cost of slightly more storage space overhead, but the underlining erasure code used as component for these codes are still basically the Reed-Solomon or an XOR-based erasure code. So it is now time again to evaluate performance of current popular implementations of open source erasure code libraries. In addition, we evaluate a public domain erasure code [11] as well, which can be readily implemented by data storage practitioners. The goal of this paper is to provide a guideline for data storage community to choose suitable erasure code for their systems and applications, especially those with high performance needs and requirements.

This paper is organized as follows: Section II introduces basics of erasure codes and related work; Section III describes preliminary experimental results for performance evaluation; detailed performance evaluations of encoding and decoding are presented in Section IV and V respectively; Section VI concludes the paper.
II. ERASURE CODES AND RELATED WORK

Erasure codes are mathematical transformations to provide reliability for various data storage systems [18] [24]. For an \((n, k, m)\) erasure code, an original message or data consists of \(k\) equal size symbols, then \(m\) parity symbols are computed from the \(k\) data symbols, through an encoding operation. The \(k\) data symbols and \(m\) parity symbols together form a codeword of \(n\) symbols, where \(n = k + m\), such that loss or erasure of any \(e\) symbols can be tolerated, i.e., the original \(k\) data symbols can still be exactly recovered from the surviving \(n-e\) symbols through a decoding operation. Obviously, by simple the pigeonhole principle, \(n-e \geq k\), i.e., \(e \leq m\). When \(e = m\), such an erasure code is called the Maximum Distance Separable or simply MDS code [18]. An MDS erasure code is optimal in terms of space efficiency for a designed erasure recovery degree \((e)\), and thus most desired in many systems and applications, including data storage systems. All erasure codes discussed in this paper are MDS codes. Note again that an LRC or a SDC code just employs an MDS code as a component code.

Now we define erasure code related nomenclature in the context of data storage system, which will be used throughout this paper. A data storage system is composed of an array of \(n\) disks in total. Each individual disk has the same size. These \(n\) disks are partitioned into two categories: \(k\) of them contain the original data, and \(m\) of them contain the redundant coding data that is calculated from the original data. We call the first category the data disks, while the second category the parity disks. We label the data disks \(D_0, D_1, ..., D_{k-1}\) and the parity disks \(C_0, C_1, ..., C_{m-1}\). An erasure code for such a system is represented as a \((k, m)\)-code. Obviously, we have \(n = m + k\). Such a typical system can be described as in Figure 1.

![Fig. 1: A typical storage system with erasure coding.](image)

The encoding operation, where the content of \(k\) parity disks are computed from those of the \(m\) data disks, partitions each disk into several strips (blocks or symbols) of a certain size, called blocksize. When an encoding/decoding operation is performed, one strip will be used from each disk. All together, \(n\) \((n = k + m)\) strips will be used. This group of \(n\) strips is called a stripe or codeword. Thus, the whole storage system is an array consisting of \(n\) disks. Each stripe is a sub-array consisting of \(n\) strips. Here each disk is represented as a column in the array.

When encoding and decoding operations are performed, each strip is partitioned into \(r\) rows. This \(r\) is usually decided by erasure code algorithm employed. For each strip, each row is simply an operation unit of a packet. Its size is called packetsize, thus blocksize = packetsize * \(r\). Each stripe is encoded and decoded independently, so that load balancing can be achieved by performing rotating and switching the disks’ identities for each stripe. It is easy to see that in a distributed system, each disk can be just a single node. But throughout this paper, we stick to the term disk.

As already mentioned, this paper focuses on the MDS codes, where loss of any up to \(m\) disks can be tolerated. Over the years, quite some MDS codes have been designed and implemented. Based on the basic computation employed in encoding and decoding operations, they in general belong to two classes: 1) finite field operations are needed; 2) only simple binary XORs (exclusive-OR) are needed. This first class is represented by the most versatile and powerful Reed-Solomon code [26]. Codes in the second class are often called the array codes, examples of which include the EVENODD code [4] and its generalizations [5], the X-Code [29], the RDP code [7], and the STAR code [11] and generalized RDP code [3]. Finite field operations are often more expensive than simple binary XORs, but erasure codes in the first class can have more flexible choice of \((k, m)\), whereas array codes in the second class so far only have limited choice of \(m\). For example, \(m = 2\) for the EVENODD code, X-Code and RDP code, and \(m = 3\) for the STAR code and generalized RDP code [3].

There exist various implementations of erasure codes and this paper does not intend to repeat good survey results in [24], thus only focuses on results that will supplement those in [24]. Due to its versatility and long history, not surprisingly, Reed-Solomon code has been employed in most data storage systems for \(m \geq 3\) either directly or as a component code for LRC type erasure codes.

A. Reed-Solomon code

Reed-Solomon (RS) Code dates back to the 1960s [26]. Original RS code was described in polynomial form, but now most of its implementations adopt matrix form for easy understanding and implementation to be used as erasure codes. (RS codes are much more than just erasure codes, more importantly, they can correct errors in various communication and storage systems [15]). Using our terms described above, RS Code assumes that each codeword packet, i.e., packet in a strip (block), is a \(w\)-bit word and \(r = 1\). Here \(w\) must satisfy \(n \leq 2^w + 1\). Usually, \(w \in \{8, 16, 32, 64\}\), and is decided by
the user, as long as it satisfies \( n \leq 2^w + 1 \). Smaller \( w \) requires less computation thus yields better performance. In most use cases \( w = 8 \) is sufficient to meet most system needs of \( n \). Each packet in Reed-Solomon code is treated as a number between 0 and \( 2^w - 1 \), and these numbers are operated in a Finite Field or Galois Field (GF(\( 2^w \))). Galois Field arithmetic is a closed and well-behaved system, in which addition, multiplication and division are defined.

Encoding of Reed-Solomon code is simply linear algebra. A Generator Matrix (\( G^T \)) is constructed from a Vandermonde Matrix. \( G^T \) is then multiplied by the \( k \) data strips (blocks) to create a codeword, consisting of \( k \) data and \( m \) parity strips (blocks). This process is illustrated as in Figure 2 where \( k = 4 \) and \( m = 2 \).

![Fig. 2: Reed-Solomon code encoding process.](image)

When disk erasures (failures) occur, the decoding process is equivalent to solving a set of independent linear equations, by deleting rows of \( G^T \), inverting it, and multiplying the inverse by the surviving blocks. Since \( G^T \) is constructed from the Vandermonde Matrix, it is ensured that the matrix inversion is always successful.

The disadvantage of Reed-Solomon code is that, in Galois Field Arithmetic, while addition is equivalent to bitwise exclusive-or (XOR) \([17]\), multiplication is more complicated, typically implemented with multiplication tables or discrete logarithm tables \( [9] \). This makes Reed-Solomon code computationally expensive.

One development since performance evaluation of Reed-Solomon code in \([24]\) is that Intel’s SSE \([12]\) has included fast multiplication operations for finite field by using parallel multiplication table lookups and thus improving multiplication speeds significantly \([23]\). Both Jerasure 2.0 \([22]\) and Intel’s ISA-L \([13]\) have adopted this speedup technology, which will be focuses of this paper. In both libraries, basic finite field multiplications are performed over a 128-bit word instead of 8-bit word (even though \( w \) remains to be 8, i.e., the finite field used is still GF(\( 2^8 \))), hence \( \text{packetsize} = 16 \text{ Bytes.} \)

### B. STAR code

Designed in 2007 \([11]\), the STAR code belongs to array code class. It is both an alternative and an extension of the EVENODD code that was designed in 1994 \([4]\).

STAR code is an efficient erasure code using only XOR operations. STAR code can tolerate up to three disk erasures \( \text{as} \) introduced before, Erasure Code consists of \( k \) data disks and \( m \) parity disks. For STAR code, \( m = 3, k \leq p, \) where \( p \) is a prime number and \( r = p - 1 \) as shown in Figure 1.

Performance evaluation of other array codes of \( m = 2 \), such as the EVENODD code and the RDP code, was conducted and presented in \([24]\), but practical performance of STAR code has not been published. Thus this paper will use STAR code as a representative of array codes for performance study, for the reasons: 1) EVENODD code is a just a special case of STAR code for \( m = 2 \), in fact decoding performance of STAR code for recovering from 1 or 2 disk erasures well represents that of EVENODD code; 2) encoding and decoding performance of \( m = 3 \) has more meaningful guidance for modern storage systems that need higher reliability degree.

The general structure of STAR code is very similar to the EVENODD code. On top of EVENODD code, STAR code adds one additional parity column. The encoding complexities averaged over parity data of EVENODD and STAR codes are the same, but the decoding complexity of STAR code is more optimized. XOR is the only operation that is used in STAR, for both encoding and decoding. Figure 3 shows a typical structure of STAR code with \( k = 5 \), and how parity column III is generated. Note that the bottom row is an imaginary row. More comprehensive description and analysis of STAR code can be found in \([11]\).

![Fig. 3: STAR code: generating parity column III.](image)

### C. Open Source Libraries

There have been a number of open source erasure coding libraries that support RS code, such as Jerasure \([22]\) and Intel’s ISA-L \([13]\) in C, BackBlaze \([2] \) in Java, Zfec \([19]\) in Python. While there is no open source library of STAR code yet, its encoding and decoding algorithms are in public domain \([11]\), unlike EVENODD code or RDP code, both of which were patented. We thus implemented our own version of STAR code in C, with some performance optimization techniques we developed \([17]\). Throughout this paper, this version of STAR code implementation is used for all tests.

For high performance, the following open source implementations of the RS code are selected for tests in this paper, as well as our implementation of STAR code:

- **Jerasure** \([22]\): Jerasure is an open source library written in C. It supports erasure coding in storage applications. A variety of erasure codes are integrated in Jerasure, including Reed-Solomon code. Reed-Solomon code may
be based on Vandermonde or Cauchy matrices. But the Vandermonde implementation has been better supported and achieves better performance by utilizing Intel’s SIMD instructions. Thus only the performance of the Vandermonde implementation of RS code is presented in this paper. A user can choose different parameters such as blocksize and finite field word size \( w \). From our tests, for a practical storage system with less than 256 disks in total for a stripe, \( w = 8 \) gives best encoding and decoding performance. Thus only test results of \( w = 8 \) are presented in this paper.

Jerasure includes a comprehensive implementation of finite field operations using Intel’s SSE, which gives a competitive performance of Reed-Solomon code among open source implementations. In order to use Jerasure Library, **GF-Complete Library** [23] must be installed first. In this paper, we use Jerasure 2.0 (together with GF-Complete 1.02, both lastest versions available) as a tool to test encoding and decoding performance of Reed-Solomon code.

- **Intel ISA-L**: Intel’s Intelligent Storage Acceleration Library (Intel ISA-L) is an open source library developed by Intel [13]. It supports not only erasure codes, but also RAID, Cyclic Redundancy Check, etc. It uses Reed-Solomon code as erasure codes. Intel ISA-L is majorly implemented in C, but some key components are implemented in assembly language to optimize performance. In this paper Intel ISA-L v2.14.1 is used to test encoding and decoding performance of Reed-Solomon code.

- **STAR code**: As shown later in Sec III-B, Intel’s SSE instruction of 128-bit XOR does bring significant performance improvement for STAR code as well, thus implementation of STAR code in this paper does utilize SSE speedup too.

### III. Experiment Setup and Baseline Measurement for Performance Evaluation

First we describe basic system setup for all experiments in this paper. All tests are conducted on a Lenovo Thinkcentre M900 equipped with memory of 4GB, and a CPU of Intel’s i5-6500, which has 256KB of L-1 cache, 1024KB of L-2 cache and 6MB of L-3 cache. The OS is Ubuntu 16.04.3 LTS, with gcc version of 5.4.0. When compiling, gcc uses option \(-O3\) consistently for all test programs.

For all experiments, data file is split and encoded into \( n = k + m \) pieces. Each piece is stored on a separated disk, so that the system tolerates up to \( m \) disk erasures. In other words, the encoder of all libraries will read a data file, encode it, and write it to \( k + m \) data/parity files, while the decoder will read the \( k + m \) data/parity files, and reconstruct the original file.

For the same reason as in other tests [24], [17], [23], all test operations in this paper are performed in memory with no actual disk I/O involved so that encoding and decoding performance can be assessed more accurately. Thus for the rest of this paper, a file really refers to data already in memory. We adopt exact same method in [24] for measuring time and calculating encoding and decoding speeds.

#### A. Baseline Performance

![Fig. 4: memcpy and XOR throughputs of the testing machine](image)

The speeds of basic `memcpy` and `XOR` are used to represent the baseline performance of the testing machine. The results are shown in Figure 4. Note that for the baseline tests, the `blocksize` is set to 1KB or 2KB and the testing data size is 1000 stripes (codewords). (Throughout this paper, all test results are measured by averaging over 1000 stripes to mitigate fluctuations in individual test result.) The x-axis represents the number of data disks \( k \), starting from 6 to 17 to reflect configurations in usual practical storage systems, while the y-axis represents the throughput in GB/s. The average speeds of `memcpy` and `XOR` are very close to each other, while `memcpy` is slightly faster than `XOR`, which is not surprising. Also note that performance is better with blocksize of 2KB than 1KB for both `memcpy` and `XOR` because of better use of caches.

#### B. Impact of Intel’s SSE

Now we examine impact of Intel’s SSE to performance of encoding and decoding. Intel’s Streaming SIMD Extensions (SSE) has been widely integrated in modern microprocessors of many brands [28]. Intel’s SSE provides eight 128-bit general-purpose registers, each of which can be directly addressed using the register names XMM0 to XMM7. Each register consists of four 32-bit single precision, floating-point
numbers, numbered from 0 through 3. As shown in Figure 5 when performing XORs, Intel’s SSE performs the SIMD XOR of the four-packed single precision floating-point values from the source operand and the destination operand, and then stores the packed single precision floating-point results in the destination operand. The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register.

![Fig. 5: Intel SSE XOR xmm0, xmm1.](image)

Intel SSE instructions are already integrated in both Jerasure and ISA-L to accelerate Finite Field multiplication speed, and thus encoding and decoding performance of RS code. They can also be used to speed up XORs used in STAR code:

- _m128 dst = _mm_load_ps (float const* mem_addr) loads 128 bits (composed of 4 packed single precision (32-bit) floating-point elements) from memory into dst. mem_addr must be aligned on a 16-byte boundary.
- void _mm_store_ps (float* mem_addr, _m128 a) stores 128 bits (composed of 4 packed single precision (32-bit) floating-point elements) from a into memory. mem_addr must be aligned on a 16-byte boundary.
- _m128 dst = _mm_xor_ps (_m128 a, _m128 b) computes the bitwise XOR of 4 packed single precision (32-bit) floating-point elements a and b, and store the results in dst.

In order to examine performance improvement brought by SSE, two different versions of STAR code have been implemented and compared: one without using Intel’s SSE and the other using Intel’s SSE. In the tests k ranges from 6 to 17, and blocksize is set to 1KB and 2KB. Test results are shown in Figure 6 with x-axis representing k and y-axis representing the encoding/decoding speed in GBs/sec. Decoding is for recovering 3 erasures. Again, results are not surprising: by applying Intel’s SSE, STAR code’s encoding/decoding speeds are indeed improved by about 40% to 80%, averaging more than 50%. Thus for rest of this paper, only results of STAR code’s SSE implementation will be presented.

![Fig. 6: SSE’s impact on STAR code performance: encoding and decoding for 3 erasures](image)

IV. ENCODING PERFORMANCE EVALUATION

Now we study practical encoding performance of RS codes and STAR codes. As already shown in [24], many factors greatly affect performance of both Reed-Solomon codes and STAR codes, such as the number of data columns k, the number of parity columns m, encoding block size blocksize, size of total data to be encoded/decoded, cache sizes of the testing machine and activities of other applications. Thus all measurement results shown in this paper can only represent a general guidance or indication instead of accurate predictions in production systems. In order to make maximum use of all levels of caches (L-1 to L-3) and make all measurements more smooth, as already noted enough iterations (1000 codewords or stripes) are run and results are averaged. Note for fair comparisons, m=3 for all tests; and for best performance, w = 8 in both Jerasure and ISA-L for Reed-Solomon code implementations, with 128-bit GF(2^8) multiplication using Intel’s SSE. Adopting the same experimental methodology in [24], we study effects of various parameters on encoding performance and decoding performance as well in later section V.

A. Impact of blocksize

For practical storage applications and systems, k and m are usually decided by needs or requirements, leaving little room to change to tune encoding or decoding performance for an
erasure code. Hence the most important parameter affecting encoding/decoding performance is blocksize.

As described in Section II, block is the basic operation unit for encoding and decoding computation on each disk, and files are split into blocks before being passed to erasure coding library. From Figure 1, blocksize = packetsize \times r. For RS code in Jerasure and ISA-L, packetsize = 16 (Bytes), and r is thus decided by blocksize specified by user/application: r = \left\lceil \frac{\text{blocksize}}{16} \right\rceil. For STAR code, XORs are performed using Intel’s SSE over 128-bit word (16 Bytes), but r = p - 1, where p has to be a prime no less than k. For best performance, p needs to be as small as possible. Hence given k and blocksize, p is first chosen to the smallest prime number no less than k and then packetsize is decided by packetsize = 16 \times \left\lceil \frac{\text{blocksize}}{16 \times r} \right\rceil). For both codes, the real coding block size is then computed as blocksize = packetsize \times r, i.e., not necessary exactly same as the blocksize specified by user/application, but fairly close.

In most practical storage systems or applications, though, the blocksize is preferably to be in the form of 2^b (where b is a positive integer) bytes, such as 1KB, 2KB, 4KB or even 8KB. This is easily achievable for the RS code, as packetsize = 16 = 2^4 (bytes).

On the other hand, for STAR code, if r is not in the form of 2^n, blocksize cannot be in the form of 2^b bytes, even though packetsize can always and should be chosen to be in the form of 2^n bytes. Recall r = p - 1 for STAR code, where p is a prime number no less than k. Fortunately, p = 17 is a prime with corresponding r = 16 = 2^4. This choice of p can support a STAR code with k (number of data disks) up to 17, which can meet needs of most systems and applications. For larger systems and applications where k needs to be larger than 17, p can then be chosen to 257 with r = 256 = 2^8. When r is too large compared to k, encoding/decoding performance will degrade a bit, as will be shown later. Thus in most of following tests in this paper, when k \leq 17, p is chosen to be 17 and r = 16 for STAR code, with packetsize = blocksize/16.

To illustrate impact of blocksize on encoding performance, test results for k = 10 are shown in Figure 7 with blocksize ranging from 1KB to 32KB. Such a k is just to reflect usual application or system setting. These results show that both Jerasure and STAR see higher encoding throughput as blocksize increases, this is due to better L-2 and L-3 cache uses for either finite field multiplication or XOR. On the other hand, ISA-L’s encoding performance remains relative stable as blocksize increases, it is perhaps because ISA-L’s finite field multiplication is implemented in assembly language and already optimized using different levels of caches, especially L-1 cache.

As of performance comparison, ISA-L is obviously a better implementation than Jerasure, which is not surprising, as ISA-L is improved upon Jerasure. On the other hand, with SSE help for both finite field multiplication and XOR, XOR is still significantly faster than finite field multiplication, and thus STAR code enjoys higher encoding throughout than ISA-L, especially when blocksize increases.

Thus from encoding performance point of view, if use of RS code is a requirement, then ISA-L is a much better choice than Jerasure; otherwise, STAR is preferable to ISA-L. The weakness of STAR and other known array codes, is limitation of m, which can only be 3 to support reliability of tolerating up to disk failures at the same time. If more failures need to be supported, RS code has to be used.

V. DECODING PERFORMANCE EVALUATION

In this section, we compare the decoding performance of the three erasure code implementations. As discussed in II-B, STAR code can only tolerate up to 3 disk erasures. Apparently, as the number of disk erasures (m) differs, the decoding performance changes as well. It is natural to assume that it takes more time to decode more erasures. In this section, we intend to take this factor into consideration and better understand how coding parameters, such as blocksize
and $k$, affect general decoding performance. Decoding performances for disk erasure ($m = 1$), two erasures ($m = 2$), and three erasures ($m = 3$) are to be examined respectively.

### A. Impact of blocksize

Just as in Sec. [IV-A], $k = 10$ and decoding throughputs are measured and averaged over 1000 codewords (stripes). blocksize is chosen to be 1KB, 2KB and 4KB too. The decoding performance results are shown in Figure 9a, 9b, 9c for $m = 1, 2, 3$, respectively. The x-axis represents packetsize while y-axis represents decoding speed in terms of GBs/sec.

From these results, we observe

1) Just like encoding, ISA-L’s decoding performances remain relatively stable as blocksize changes for all $k$’s and erasure numbers, for the similar reason;

2) Again like encoding, decoding performances of both Jerasure and STAR increase as $k$ increases, for all $k$’s and erasure numbers, and for the similar reason explained in encoding. After all, decoding employs similar operations (finite field multiplication and XOR) in encoding, besides decoding paths (indices of packets to multiply or XOR);

3) When decoding 1 erasure, ISA-L outperforms Jerasure as blocksize increases, with 4KB as a cross-point in these results. But more importantly, STAR’s throughputs are much higher than those of Jerasure and ISA-L, by a factor of roughly 1.6x to 5.5x against Jerasure and about 2.2x to 2.6x against ISA-L. The reason is even though decoding 1 erasure is just like encoding for all the three codes, employing XORs only, STAR’s XOR packet size is much bigger (blocksize$/16$ bytes, varying
from 64 to 256 bytes) than that of ISA-L and Jerasure (which is 16 bytes), making much better use of L-2 and L-3 caches. Since one erasure occurs more frequently than two or three erasures in general, performance for decoding 1 erasure is a more important consideration in most storage applications and systems (in fact, this is a very reason LRC or SDC codes are designed), as is, STAR code is more preferable to Jerasure or ISA-L in decoding 1 erasure. If RS code, thus Jerasure or ISA-L, has to be used, their implementations of decoding 1 erasure need to be modified to use larger packet size to achieve much better performance as STAR;
4) When decoding multiple erasures, ISA-L’s throughput is constantly higher than that of Jerasure, while their performance gap becomes closer as blocksize increases; STAR’s decoding performance, on the other hand, remains to be highest (except very close to ISA-L’s for blocksize of 1KB when decoding 3 erasures), especially as blocksize increases. STAR outperforms Jerasure by a factor of about 2x to 3x, and ISA-L by a factor of about 1.25x to 2.1x except for blocksize of 1KB.

B. Impact of k

As in Sec. IV-B, we also measure how decoding throughput changes as k varies so that storage practitioners can have planning when disk failures occur. Again, blocksize is set to 1KB, 2KB and 4KB respectively, and k varies from 6 to 17, and p = 17 for STAR code.

Figure 10, Figure 11 and Figure 12 respectively show performance of decoding one erasure, two and three erasures, where again The x-axis represents k while y-axis represents decoding speed in terms of GBs/sec.

The above results show
1) As shown in Sec. IV-B for all the three block sizes, Jerasure’s encoding performance seems to be most stable against k, albeit at much lower throughput than that of either ISA-L or STAR;
2) Again ISA-L and STAR, on the other hand, do exhibit fluctuations in decoding performance as k changes, though not dramatically;
3) Also consistent to Figure 9 STAR performs much better than both ISA-L and Jerasure when decoding 1 erasure, for all blocksize, for the reason already discussed;
4) When blocksize = 1KB, for all k’s, decoding performances of STAR and ISA-L are quite close for multiple erasures (m = 2 or 3), but much higher than that of Jerasure;
5) When blocksize ≥ 1KB, STAR’s decoding performance is highest among the three for all k’s, while Jerasure’s is lowest. The performance gap between STAR and ISA-L as blocksize increase.

C. p = 17 vs. p = 257 for STAR

Finally recall that r = p − 1 for STAR code, where p is a prime number that is no less than designed k. In order to make blocksize to be in form of 2^b, p = 17 or 257 is a reasonable choice to achieve the goal. For a given blocksize, packetsize = blocksize/r. So when r = 256, its corresponding packetsize is just 1/16 of that when r = 17, e.g., when blocksize = 2KB, packetsize = 8 bytes for p = 257, but packetsize = 256 bytes for p = 17. A larger packetsize at this order can usually make better use of L-1 and L-2 caches for XORs, and thus decoding and encoding performance. The following experimental results do verify this behavior.

Figure 13, Figure 14 and Figure 15 display encoding and decoding performances of STAR code with p = 17 vs. p = 257 for blocksize of 1KB, 2KB and 4KB respectively. These results demonstrate that p = 17 constantly yields about 10% to 15% better encoding and decoding throughput than p = 257 for all k’s and blocksize, and for any erasure numbers. This indicates p should be set to 17 whenever possible, i.e., for all k ≤ 17.
VI. CONCLUSIONS

While the purpose of this paper is to present raw performance measurement data to data storage practitioners and researchers, and let the data speak for themselves so that data storage practitioners and researchers can make their own conclusions according to their needs and requirements, we can still make the following general observations:

1) **Intel’s SSE**: SSE instructions do greatly help improve both encoding and decoding performances of all the three coding libraries, and all the three coding implementations reach GBs/sec on the testing machine with quite common configurations. But further performance improvement beyond general hardware instruction set assistance needs to come from better use of all levels of caches and algorithmic designs and implementations of coding libraries;

2) **Jerasure vs. ISA-L**: while both libraries are good open source implementations of RS code, extensive performance measurement data shows ISA-L performs much better in encoding operations for all \((k, blocksize)\) combinations and also meaningfully better in decoding operations for most \((k, blocksize)\) combinations, except when decoding one erasure for large block sizes (larger than 4KB in our tests). Thus for most use cases, ISA-L is a preferable implementation for RS code;

3) **STAR vs. RS code**: thanks to the fact that XORs are still more efficient than finite field multiplications even with SSE assistance, STAR code always exhibits higher encoding throughputs for all parameters, especially for large block sizes; STAR code also performs better for decoding in most \((k, blocksize)\) combinations, especially when decoding one erasure. Thus whenever possible, STAR code (or other similar array code using only XORs) is preferable to RS code when \(m = 3\) is enough for system reliability;

4) **Coding block size**: For most applications and systems, coding block size is perhaps the only parameter a user can tune to change encoding and decoding performances. While ISA-L’s encoding and decoding performances are relatively stable against the coding combination.
block size, both Jerasure and STAR can have higher encoding and decoding throughputs as block size increases thanks to better use of multiple levels of caches. Hence when possible, a larger coding block size should be chosen in order to achieve better encoding and decoding performance;

5) Further Improvement of ISA-L and Jerasure: current implementations of the two libraries are using small XOR packet size (16 bytes) for all encoding and decoding operations. But when decoding one erasure, they actually only use XORs instead of finite field multiplications. To better use L-1 and L-2 caches, a larger XOR packet size should be used for much higher decoding throughput, as demonstrated in STAR code. Current implementations of ISA-L and Jerasure can thus be modified in the decoding one erasure component for much better performance. As one failure occurs much more often than multiple erasures for most data storage systems and applications, this modification will greatly benefit most use cases.
REFERENCES


