SN Computer Science (2020) 1:54
https://doi.org/10.1007/s42979-019-0057-1

ORIGINAL RESEARCH

Practical Performance Evaluation of Space Optimal Erasure Codes for High-Speed Data Storage Systems

Rui Chen[1] · **Lihao Xu**[1]

Received: 12 October 2019 / Accepted: 12 December 2019 / Published online: 23 December 2019

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 Plank et al. (Fast, 9:253–65, 2009) 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 practitioner 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/s. 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.

Keywords Erasure codes · Performance evaluation · Data storage systems

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 [9, 24], Google's File System [10] and its successor Colossus [5], Microsoft's Azure [14, 15], and Facebook's storage systems [29, 33]. 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). It has been perceived that erasure code encoding and decoding operations have not become a critical bottleneck in most data storage systems. However, 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 technologies such as flash-based solid-state drive (SSD), non-volatile memory (NVM), and 3D XPoint technology, are blurring speed boundary between main memory and persistent storage, making persistent storage IO speed easily reaching GB/s [19, 34, 36]. 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 satisfy 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. Furthermore, software-defined storage systems (SDS) also call for more efficient erasure code encoding and decoding operations to improve overall system's performance [11].

It has been almost a decade since last time performance of then-popular open-source erasure code libraries was

✉ Rui Chen
chenrui@wayne.edu

Lihao Xu
lihao@wayne.edu

[1] Wayne State University, Detroit, MI 48202, USA

evaluated [28]. Ever since then, implementations of existing erasure codes, mainly the versatile Reed–Solomon code [32], have been improved a lot, especially by employing Intel's Streaming SIMD Extensions (SSE) instruction set [8] for efficient finite field operations and XORs [27]. Another change in erasure code practice since then is the use of so-called local reconstruction codes (LRC) [14] and sector-disk (SD) codes [25] 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 is 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 [13] as well, which can be readily implemented by data storage practitioners. We choose not to include patented erasure codes, such as the RDP code [6] and its extension, since they are not widely employed in other storage systems. 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: the next section introduces the basics of erasure codes and related work, followed by which preliminary experimental results for performance evaluation are described; detailed performance evaluations of encoding and decoding are presented in the next two subsequent sections, respectively; the final section concludes the paper.

Erasure Codes and Related Work

Erasure codes are mathematical transformations to provide reliability for various data storage systems [21, 28]. 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 the simple *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 [21]. An MDS erasure code is optimal in terms of space efficiency for a designed erasure recovery degree (e) and, therefore, is most desired in many systems and applications, including data storage systems. Thus, all erasure codes discussed in this paper are MDS codes. Note again that an LRC code or an 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 are 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 $D_0, D_1, \ldots, D_{k-1}$ and the parity disks $C_0, C_1, \ldots, C_{m-1}$. 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 Fig. 1.

The encoding operation, where the content of m parity disks is computed from those of the k 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, since MDS codes are optimal in space usage, this paper focuses only 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

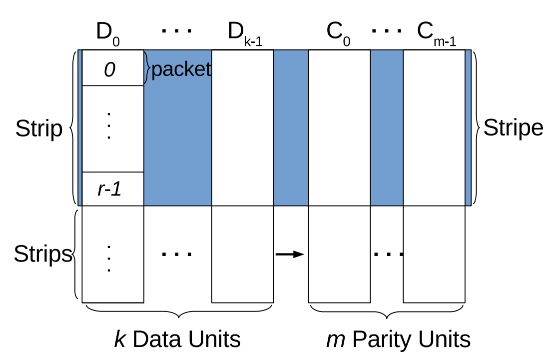


Fig. 1 A typical storage system with erasure codes

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 [32]. Codes in the second class are often called the *array codes*, examples of which include the EVENODD code [2] and its generalizations [3], the X-Code [37], the RDP code [6], and the STAR code [13] and generalized RDP code [4]. 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 [4].

There exist various implementations of erasure codes and this paper does not intend to repeat good survey results in [28], thus only focuses on results that will supplement those in [28]. Due to its versatileness 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.

Reed–Solomon code

Reed–Solomon (RS) code dates back to the 1960s [32]. 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 [18]). 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 Fig. 2, where $k = 4$ and $m = 2$.

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

$$\underbrace{\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ x_{0,0} & x_{0,1} & x_{0,2} & x_{0,3} \\ x_{1,0} & x_{1,1} & x_{1,2} & x_{1,3} \end{bmatrix}}_{G^T} * \underbrace{\begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ d_3 \end{bmatrix}}_{\text{Data}} = \underbrace{\begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ d_3 \\ c_0 \\ c_1 \end{bmatrix}}_{\text{Codeword}} \begin{matrix} \left.\vphantom{\begin{matrix}1\\1\\1\\1\end{matrix}}\right\} \text{Data} \\ \left.\vphantom{\begin{matrix}1\\1\end{matrix}}\right\} \text{Parity} \end{matrix}$$

Fig. 2 Reed–Solomon code encoding process

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) [20], multiplication is more complicated, typically implemented with multiplication tables or discrete logarithm tables [12]. This makes Reed–Solomon code computationally expensive.

One development since performance evaluation of Reed–Solomon code in [28] is that Intel's SSE [8] has included fast multiplication operations for finite field using parallel multiplication table lookups and thus improving multiplication speeds significantly [27]. Both Jerasure 2.0 [26] and Intel's ISA-L [16] 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 *packetsize* = 16 bytes.

STAR code

Designed in 2007 [13], 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 [2].

STAR code is an efficient erasure code using only XOR operations. STAR code can tolerate up to three disk erasures [13]. 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 Fig. 1.

Performance evaluation of other array codes of $m = 2$, such as the EVENODD code and the RDP code, was conducted and presented in [28], 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.

	Data columns							Parity column III
	A	E	D	C	B	X	X	A
	B	A	E	D	C	X	X	B
	C	B	A	E	D	X	X	C
	D	C	B	A	E	X	X	D
imaginary row	E	D	C	B	A	X	X	E

Fig. 3 STAR code: generating parity column III

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 [13].

Open Source Libraries

There have been a number of open source erasure coding libraries that support RS code, such as Jerasure [26] and Intel's ISA-L [16] in C, BackBlaze [1] in Java, Zfec [23] in Python. While there is no open source library of STAR code yet, its encoding and decoding algorithms are in public domain [13], 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 [20]. 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* [26] 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. To use Jerasure Library, **GF-Complete Library** [27] must be installed first. In this paper, we use Jerasure 2.0 (together with GF-Complete 1.02, both latest 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 [16]. 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 "Impact of Intel's SSE", 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.

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 two platforms, namely Lenovo Thinkcentre M900 and Intel NW200 Roke (M900 and NW200 in brief). As shown in Table 1, Lenovo ThinkCentre M900 has a CPU of Intel's i5-6500 (4 cores), which has 256 KB of L-1 cache, 1024 KB of L-2 cache and 6 MB of L-3 cache. It has a total memory of 4 GB. Ubuntu 16.04.3 LTS is installed on the machine, with *gcc* version of 5.4.0. On the other hand, Intel NW200 Roke is equipped with a CPU of Intel's Xeon E3-1275 (8 cores), which has 512 KB of L-1 cache, 2048 KB of L-2 cache and 8 MB of L-3 cache. It has a total memory of 32 GB, with OS of Ubuntu 17.10. The *gcc* version is 7.2.0. When compiling,

Table 1 Test platform configurations

Platform	CPU model	L1 cache	L2 cache	L3 cache
M900	Intel(R) Core(TM) i5-6500 @ 3.20 GHz	256 KB	1 MB	6 MB
NW200	Intel(R) Xeon(R) E3-1275 @ 3.60 GHz	512 KB	2 MB	8 MB

gcc uses $-O3$ option consistently for all testing programs. To fully use all levels of caches and provide smoother measurements, enough iterations are executed with the results being averaged.

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 [20, 27, 28], 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 the exact same method in [28] for measuring time and calculating encoding and decoding speeds.

Baseline Performance

The speeds of basic *memcpy* and *XOR* are used to represent the baseline performance of the testing platforms. The results are shown in Fig. 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 2 KB than 1 KB for both *memcpy* and *XOR* because of better use of caches.

Impact of Intel's SSE

Now we examine impact of Intel's SSE on performance of encoding and decoding. Intel's Streaming SIMD Extensions (SSE) has been widely integrated in modern microprocessors of many brands [35]. 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 [8]. As shown in Fig. 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.

Intel SSE instructions are already integrated in both Jerasure and ISA-L [16, 26] 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 four packed single-precision (32-bit) floating-point elements) from memory into *dst*. *mem_addr* must be aligned on a 16-byte boundary.

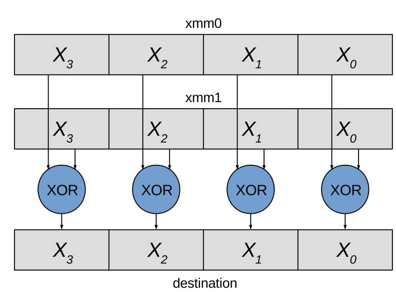
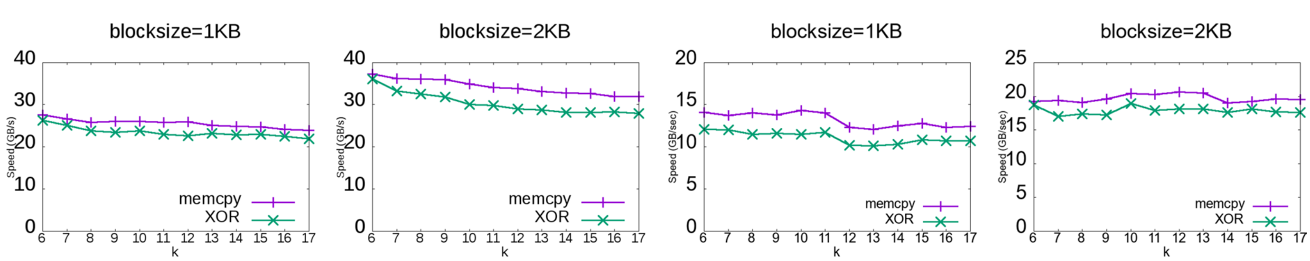


Fig. 5 Intel SSE XOR operation between xmm0 and xmm1



(a) M900 (Core i5-6500). **(b)** NW200 (Xeon E3-1275).

Fig. 4 Baseline performance of testing platforms

- *void _mm_store_ps (float* mem_addr, __m128 a)* stores 128 bits (composed of four 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 four packed single-precision (32-bit) floating-point elements *a* and *b*, and store the results in *dst*.

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 1 KB and 2 KB. Test results are shown in Fig. 6, with x-axis representing k and y-axis representing the encoding/decoding speed in GBs/s. Decoding is for recovering three erasures. Again, results are not surprising: by applying Intel's SSE, STAR code's encoding/decoding speeds are indeed improved by about 30–80%, averaging more than 50%. Thus, for the rest of this paper, only results of STAR code's SSE implementation will be presented.

Encoding Performance Evaluation

Now we study practical encoding performance of RS codes and STAR codes. As already shown in [28], 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. 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 [28], we study effects of various parameters on encoding performance and decoding performance as well in "Decoding Performance Evaluation".

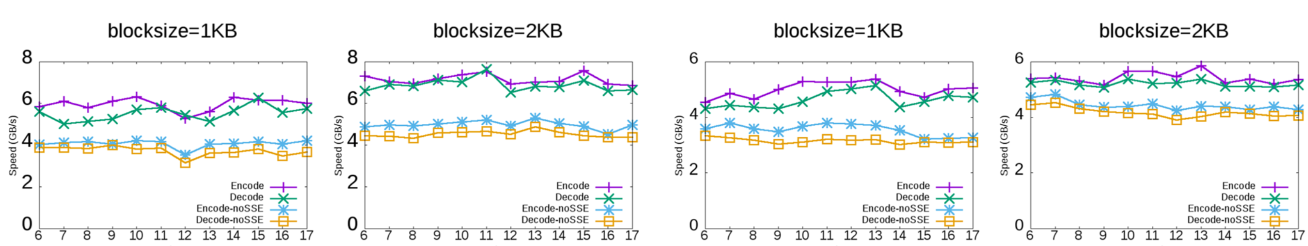
Impact of *Blocksize*

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 "Erasure Codes and Related Work", *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 Fig. 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: $r = \left\lceil \frac{blocksize}{16} \right\rceil$. 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 $packetsize = 16 \times \left\lceil \frac{blocksize}{16 \times r} \right\rceil$. For both codes, the real coding block size is then computed as $blocksize = packetsize \times r$, i.e., not necessarily exactly same as the *blocksize* specified by user/application, but fairly close.

In most practical storage systems or applications, though, the *blocksize* is preferable to be in the form of 2^b (where b is a positive integer) bytes, such as 1 KB, 2 KB, 4 KB or even 8 KB. 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^c, *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^d bytes. Recall $r = p - 1$ for STAR code, where



(a) M900 (Core i5-6500). **(b)** NW200 (Xeon E3-1275).

Fig. 6 SSE's impact on STAR code performance: encoding and decoding for three erasures

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 be 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 the impact of *blocksize* on encoding performance, test results for $k = 10$ are shown in Fig. 7 with *blocksize* ranging from 1 to 32 KB. Such a k is just to reflect usual application or system setting. These results on both platforms consistently 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 with better usage of L-1 cache using assembly language to achieve finite field multiplication. 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.

Impact of *k*

As already discussed, k is usually decided by application or storage system, we still like to examine how encoding performance fluctuates for different ks so that when there is room to choose a different k for an application or storage system, a proper k can be chosen for better encoding performance.

Figure 8 illustrates encoding performance as k varies from 6 to 17 (to meet most needs of most storage applications and systems) for typical encoding block sizes of 1 KB, 2 KB and 4 KB, respectively.

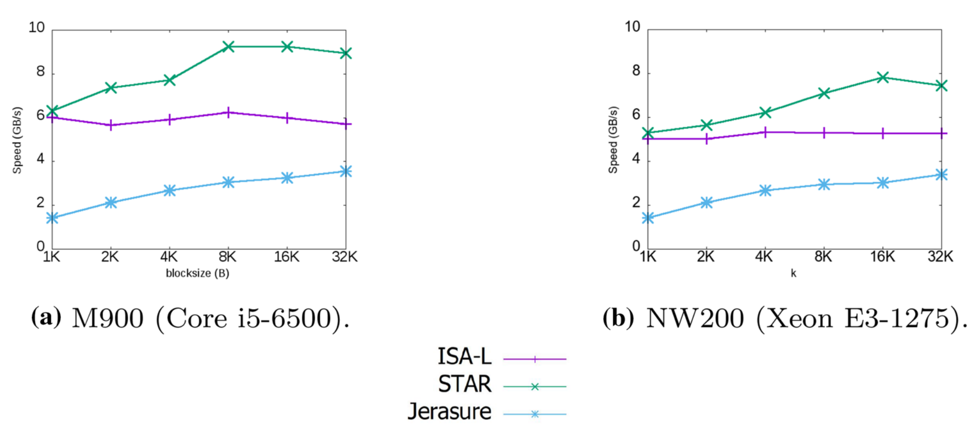
For all the three block sizes, Jerasure's encoding performance seems to be most stable against k, albeit at much lower throughput than that of either ISA-L or STAR. Both ISA-L and STAR, on the other hand, do exhibit fluctuations in encoding performance as k changes. Also consistent with Fig. 7, on both platforms, ISA-L outperforms Jerasure, while STAR performs best among the three implementations for all k and *blocksize*, though it seems STAR has slightly higher gains over ISA-L on i5 core than Xeon, while relative performance differences between ISA-L and Jerasure appear to be close on the two platforms.

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.

Decoding Performance Evaluation

In this section, we compare the decoding performance of the three erasure code implementations. As discussed in "STAR code", STAR code can only tolerate up to three 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

Fig. 7 Impact of *blocksize* on encoding performance for $k = 10$



(a) M900 (Core i5-6500). **(b)** NW200 (Xeon E3-1275).

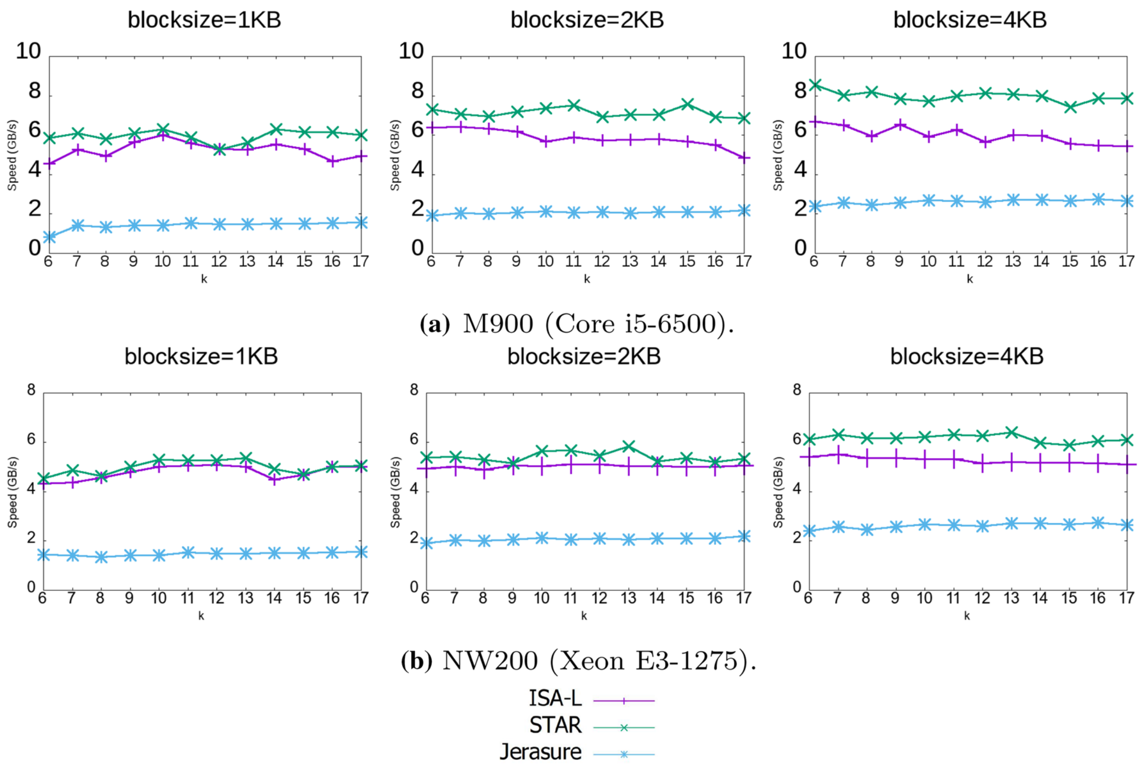


Fig. 8 Impact of k on encoding performance

($m = 1$), two erasures ($m = 2$), and three erasures ($m = 3$) are to be examined, respectively. For each set of experiments, erasure disks are randomly chosen since different erasure locations lead to different decoding speeds obviously. Enough iterations are conducted and the results are averaged to indicate a general decoding performance.

Impact of *Blocksize*

Just as in "Impact of Blocksize", $k = 10$ 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 Figs. 9a–c, 10a–c for $m = 1, 2, 3$, respectively. The x-axis represents *packetsize* while y-axis represents decoding speed in terms of GBs/s.

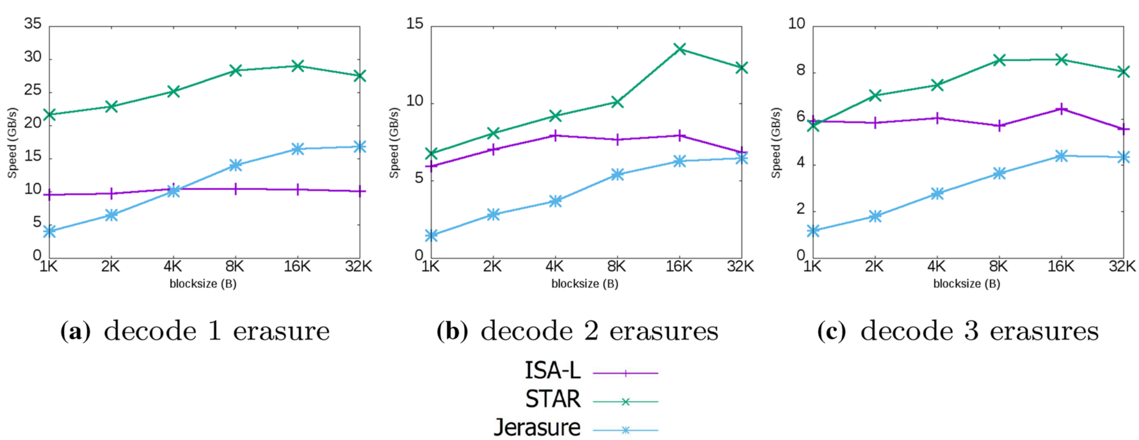


Fig. 9 Impact of *blocksize* on decoding performance for $k = 10$ on M900 (Core i5-6500)

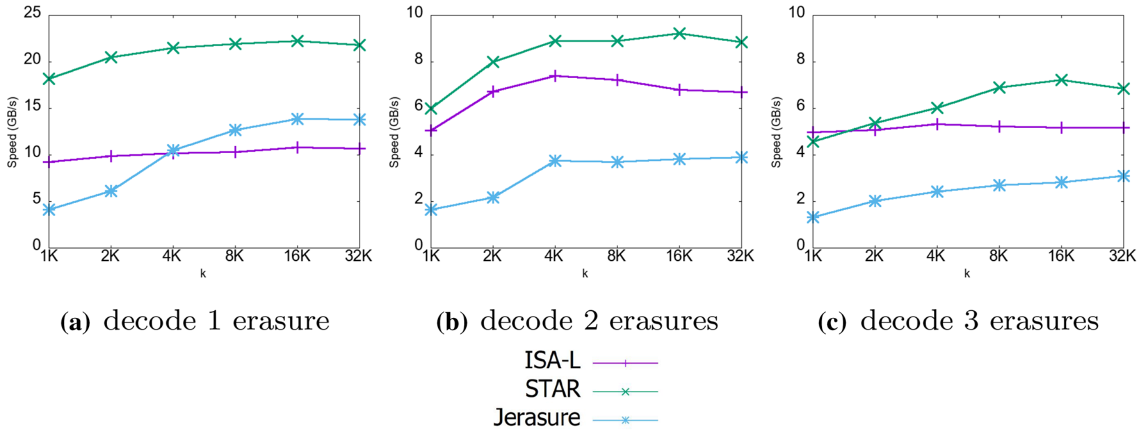


Fig. 10 Impact of *blocksize* on decoding performance for $k = 10$ on NW200 (Xeon E3-1275)

From these results, we observe on both testing platforms (i5 and Xeon):

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 4 KB 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 $1.6x$–$5.5x$ against Jerasure and about $2.2x$–$2.6x$ 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 the 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 three erasures), especially as *blocksize* increases. STAR outperforms Jerasure by a factor of about $2x$–3x, and ISA-L by a factor of about $1.25x$– $2.1x$ except for *blocksize* of 1 KB.

Impact of *k*

As in "Impact of k", 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 1 KB, 2 KB and 4 KB, respectively, and k varies from 6 to 17, and $p = 17$ for STAR code.

Figures 11a–c and 12a–c, 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/s.

The above results show

1. As shown in "Impact of k", for all the three block sizes, Jerasure's decoding 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 with Fig. 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* = 1 KB, for all ks, decoding performances of STAR and ISA-L are quite close for multiple erasures ($m = 2$ or 3), but much higher than that of Jerasure.

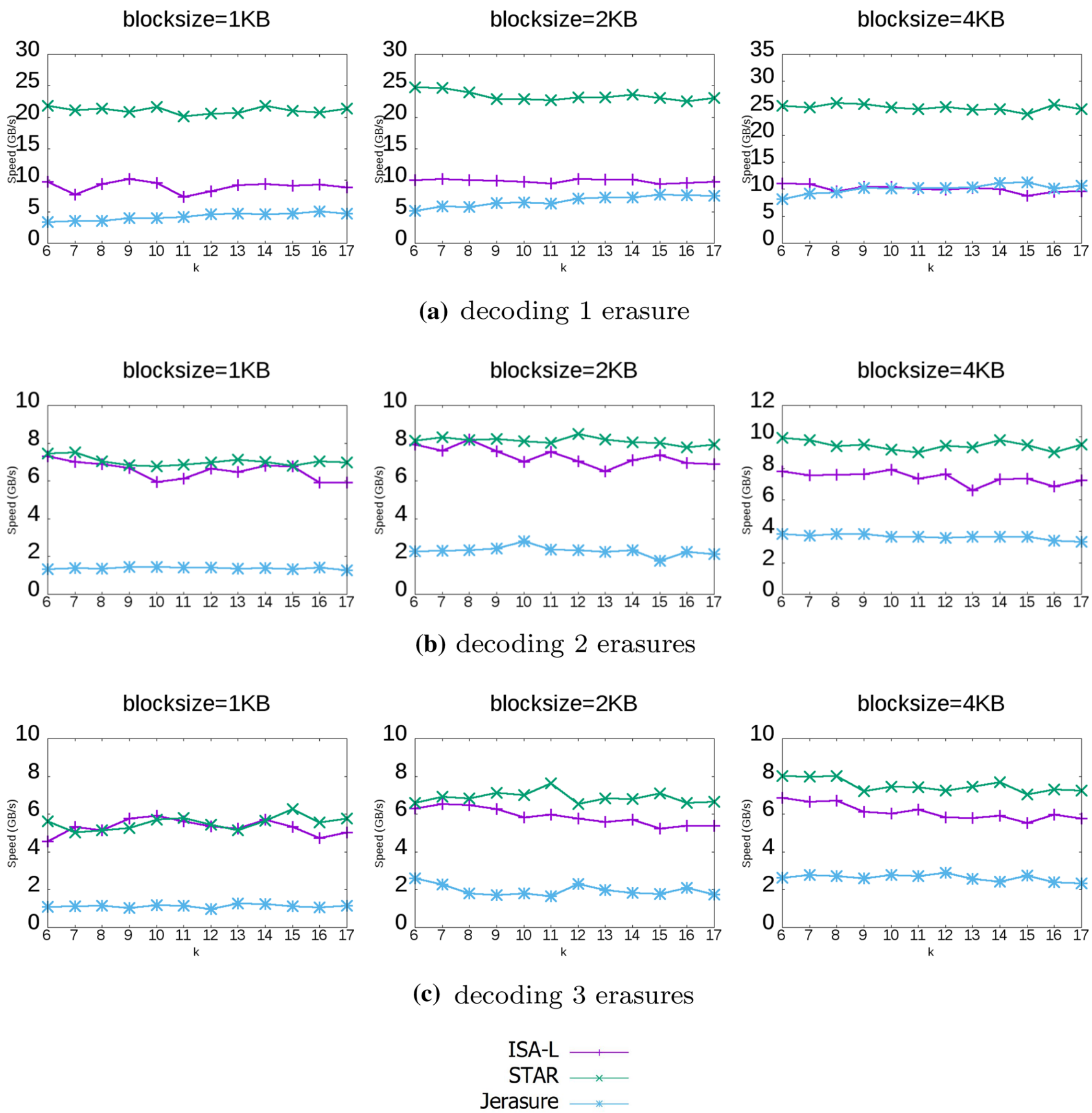


Fig. 11 Impact of k on performance for different erasures on M900 (Core i5-6500)

5. When *blocksize* > 1 KB, STAR's decoding performance is the highest among the three for all k's, while Jerasure's performance is the lowest. The performance gap between STAR and ISA-L increases as *blocksize* increases.
6. As in encoding performance, STAR has slightly better gains in decoding performance for 2 and 3 erasures, over ISA-L and Jerasure on i5 than Xeon, while relative differences between ISA-L and Jerasure remain close on the two platforms.

$p = 17$ 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. To make *blocksize* to be in the 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

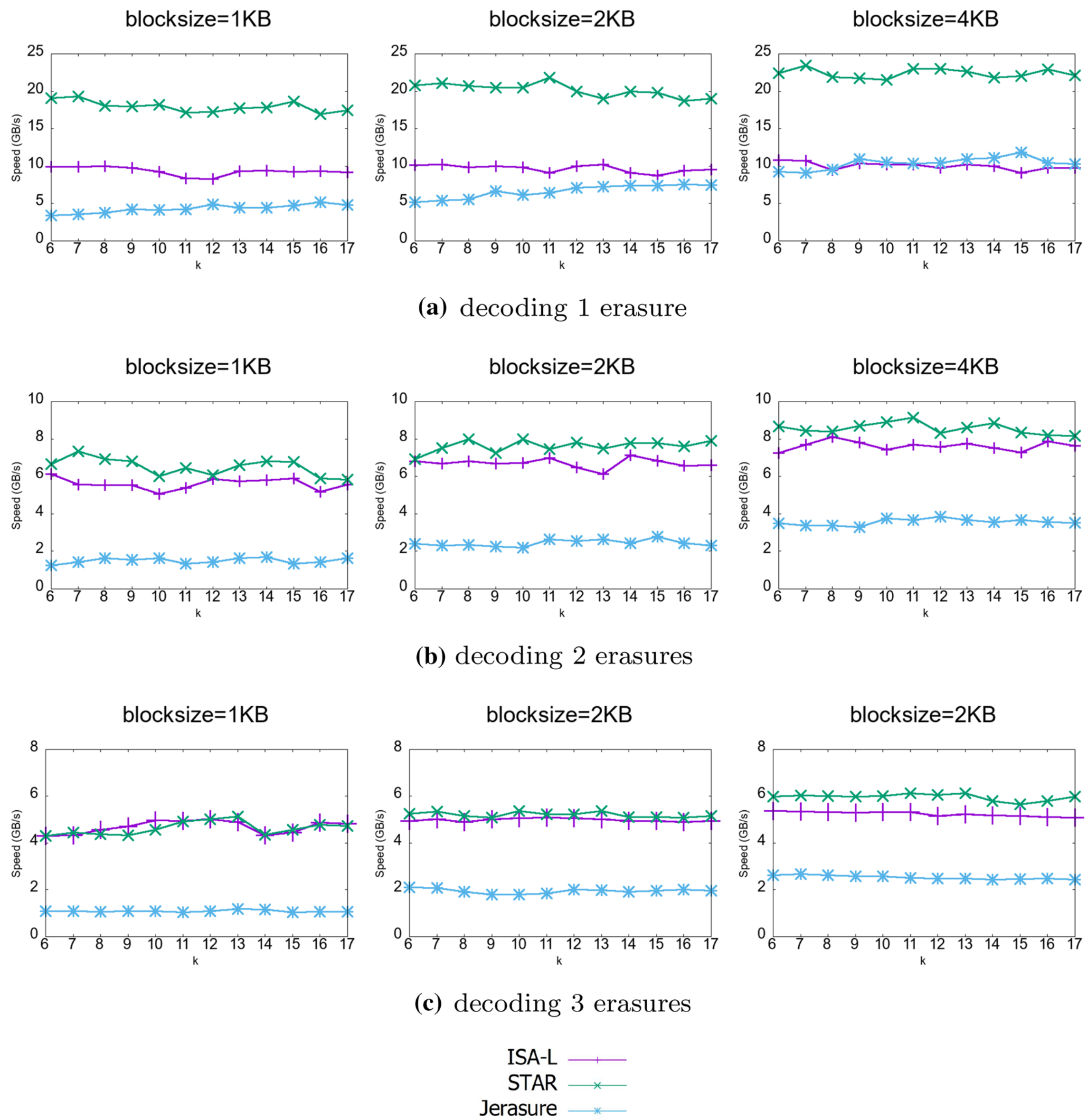


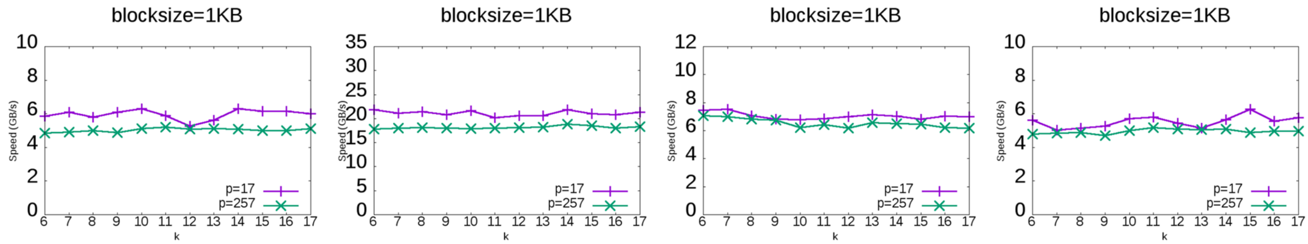
Fig. 12 Impact of k on performance for different erasures on NW200 (Xeon E3-1275)

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 on platform M900 do verify this behavior (test results on NW200 are quite similar, thus omitted here).

Figures 13, 14 and 15 display encoding and decoding performances of STAR code with $p = 17$ vs. $p = 257$ for *blocksize* of 1 KB, 2 KB and 4 KB, respectively. These results demonstrate that $p = 17$ constantly yields about 10– 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 $k \leq 17$.

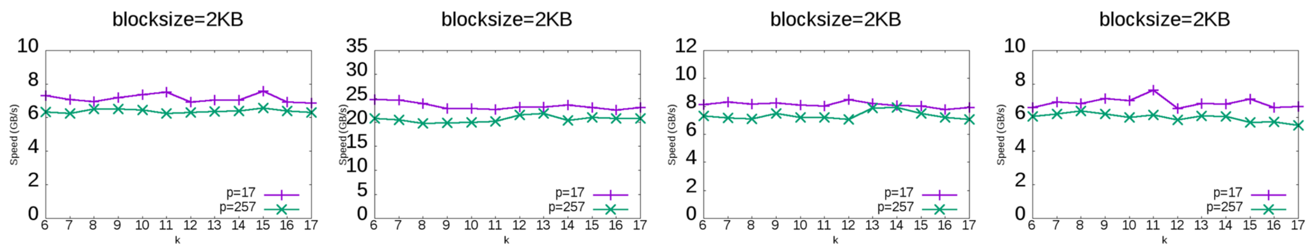
Limitations and Future Work

The purpose of this paper is to present raw performance measurement data to data storage practitioners and researchers. Thus, modifying other existing libraries, such as Jerasure or ISA-L, is beyond the scope of this paper. The evaluations are currently limited to Jerasure, Intel's ISA-L, which are typical representatives of Reed–Solomon code, and STAR code library. Due to the limitation of equipment we have in our lab, we only evaluate performance improvement brought by Intel's SSE. Data storage practitioners and researchers are welcomed to share their evaluation results with us.



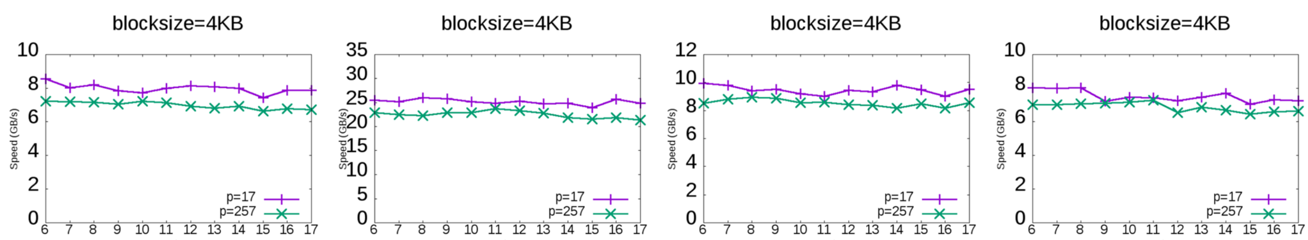
(a) Encoding **(b)** Decoding 1 **(c)** Decoding 2 **(d)** Decoding 3

Fig. 13 Encoding and decoding performance of STAR for $p = 17$ vs. $p = 257$ with *blocksize* = 1 KB



(a) Encoding **(b)** Decoding 1 **(c)** Decoding 2 **(d)** Decoding 3

Fig. 14 Encoding and decoding performance of STAR for $p = 17$ vs. $p = 257$ with *blocksize* = 2 KB



(a) Encoding **(b)** Decoding 1 **(c)** Decoding 2 **(d)** Decoding 3

Fig. 15 Encoding and decoding performance of STAR for $p = 17$ vs. $p = 257$ with *blocksize* = 4 KB

In the future, we intend to explore and benchmark more codes such as regenerating codes [30], LRC [14] and SD code [25]. We also intend to design and propose new decoding scheduling algorithm for STAR code to improve the performance.

Conclusions

While we intend to 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 following general observations:

1. *Intel's SSE* SSE instructions do greatly help improving both encoding and decoding performances of all the three coding libraries. All the three coding implementations reach GBs/s 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 4 KB 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 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 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.
6. *Different CPUs* For understandable reasons, our testbeds are only limited to the equipments we have in our lab. We understand that many data centers are running much newer CPUs, such as Intel's Cascade Lake [22] that supports AVX-512 [7], ARM with NEON technology [31], and AMD's EPYC [17]. But the main difference of coding libraries evaluated in this study and the ones in previous work [28], is the usage of Intel's SSEs, which the testbed CPUs in this study already have. As already shown in this work, the relative performances of the coding libraries are consistent on two difference testbeds. We can reasonably expect they will exhibit similar relative performances on other testbeds with different CPUs. Thus, we believe, while the testbed CPUs used in this study are relatively old, the relative performance study results will still be valuable to and provide guidelines for data storage systems with newer and future CPUs.

Compliance with Ethical Standards

Conflict of Interest Author R. Chen and author L. Xu declare that they have no conflict of interest.

References

1. Beach B. Backblaze open sources Reed–Solomon erasure coding source code. 2015. https://www.backblaze.com/blog/reed-solomon/.
2. Blaum M, Brady J, Bruck J, Menon J. Evenodd: an efficient scheme for tolerating double disk failures in raid architectures. IEEE Trans Comput. 1995;44(2):192–202.
3. Blaum M, Bruck J, Vardy A. Mds array codes with independent parity symbols. IEEE Trans Inf Theory. 1996;42(2):529–42.
4. Blaum M. A family of MDS array codes with minimal number of encoding operations. 2006. pp 2784–2788.
5. Cooper BF. Spanner: Google's globally-distributed database. In: Proceedings of the 6th international systems and storage conference. ACM; 2013.
6. Corbett P, English B, Goel A, Grcanac T, Kleiman S, Leong J, Sankar S. Row-diagonal parity for double disk failure correction. In: Proceedings of the 3th USENIX conference on file and storage technologies. USENIX; 2004.
7. Cornea M. Intel avx-512 instructions and their use in the implementation of math functions. Intel Corp. 2015.
8. Dylan B (Intel). Intel® c++ compiler 17.0 developer guide and reference. 2016. https://software.intel.com/en-us/intel-cplusplus-compiler-17.0-user-and-reference-guide.
9. EC Amazon. Amazon web services. 2015. http://aws.amazon.com/es/ec2/. Accessed Nov 2012.
10. Ghemawat S, Gobioff H, Leung S-T. The Google file system. vol. 37. ACM; 2003.
11. Goldenberg D. Erasure code offload for distributed software defined storage. In: Flash memory summit. 2017.
12. Greenan KM, Miller EL, Schwarz SJ, Thomas JE. Optimizing galois field arithmetic for diverse processor architectures and applications. In: Modeling, analysis and simulation of computers and telecommunication systems, 2008. MASCOTS 2008. IEEE International Symposium on. IEEE; 2008. pp. 1–10.
13. Huang C, Lihao X. Star: an efficient coding scheme for correcting triple storage node failures. IEEE Trans Comput. 2008;57(7):889–901.
14. Huang C, Simitci H, Xu Y, Ogus A, Calder B, Gopalan P, Li J, Yekhanin S. Erasure coding in windows azure storage. In: Proceedings of 2012 USENIX annual technial conference, ATC'12. USENIX; 2012.
15. Khan O, Burns RC, Plank JS, Pierce W, Huang C. Rethinking erasure codes for cloud file systems: minimizing i/o for recovery and degraded reads. In: FAST; 2012, p. 20.
16. Le T (Intel). Optimizing storage solutions using the intel® intelligent storage acceleration library. 2014. https://software.intel.com/en-us/articles/optimizing-storage-solutions-using-the-intel-intelligent-storage-acceleration-library.

17. Lepak K, Talbot G, White S, Beck N, Naffziger S, et al. The next generation amd enterprise server product architecture. In: IEEE hot chips. vol. 29. 2017.
18. Lin S, Costello DJ. Error control coding. 2nd ed. London: Pearson; 2004.
19. Liu W, Wu K, Liu J, Chen F, Li D. Performance evaluation and modeling of hpc i/o on non-volatile memory. In: Proceedings of IEEE international conference on networking, architecture, and storage (NAS). IEEE; 2017. pp. 1–10.
20. Luo J, Xu L, Plank JS. An efficient XOR-scheduling algorithm for erasure codes encoding. In: Dependable systems and networks, 2009. DSN'09. IEEE/IFIP International Conference on. IEEE; 2009. pp. 504–513.
21. MacWilliams FJ, Sloane NJA. The theory of error-correcting codes. Amsterdam: Elsevier; 1977.
22. Mohamed A, Bahaa F, Sailesh K, Akhilesh K, Lily P L, Sreenivas M, Andy R, Ian M S, Bob V, Geetha V, et al. Cascade lake: next generation intel xeon scalable processor. IEEE Micro. 2019;39(2):29–36.
23. O'Whielacronx Z. Zfec 1.4.24. open source code distribution. 2012. https://pypi.python.org/pypi/zfec/.
24. Palankar MR, Iamnitchi A, Ripeanu M, Garfinkel S. Amazon s3 for science grids: a viable solution? In: Proceedings of the 2008 international workshop on Data-aware distributed computing. ACM; 2008. pp. 55–64.
25. Plank JS, Blaum M, Hafner JL. Sd codes: erasure codes designed for how storage systems really fail. In: Proceedings of 11th USENIX conference on file and storage technologies, FAST'13. USENIX; 2013.
26. Plank JS, Greenan KM. Jerasure: a library in c facilitating erasure coding for storage applications—version 2.0. Technical report, Technical Report UT-EECS-14-721, University of Tennessee; 2014.
27. Plank JS, Greenan KM, Miller EL. Screaming fast galois field arithmetic using intel SIMD instructions. In: Proceedings of 11th USENIX conference on file and storage technologies, FAST'13. USENIX; 2013.
28. Plank JS, Luo J, Schuman CD, Xu L, Wilcox-O'Hearn Z, et al. A performance evaluation and examination of open-source erasure coding libraries for storage. Fast. 2009;9:253–65.
29. Rashmi KV, Shah NB, Gu D, Kuang H, Borthakur D, Ramchandran K. A "hitchhiker's" guide to fast and efficient data reconstruction in erasure-coded data centers. In: Proceedings of the 2014 ACM conference on SIGCOMM. ACM; 2014.
30. Rashmi KV, Shah NB, Ramchandran K, Kumar PV. Regenerating codes for errors and erasures in distributed storage. In: 2012 IEEE international symposium on information theory proceedings. IEEE; 2012. pp. 1202–1206.
31. Reddy VG. Neon technology introduction. ARM Corp. 2008;4(1)
32. Reed IS, Solomon G. Polynomial codes over certain finite fields. J Soc Ind Appl Math. 1960;8(2):300–4.
33. Sathiamoorthy M, Asteris M, Papailiopoulos D, Dimakis AG, Vadali R, Chen S, Borthakur D. Xoring elephants: novel erasure codes for big data. In: Proceedings of the VLDB endowment. vol. 6; 2013.
34. Suzuki K, Swanson S. A survey of trends in non-volatile memory technologies: 2000–2014. In: Memory workshop (IMW), 2015 IEEE international. IEEE; 2015. pp. 1–4.
35. Thakkur S, Thomas H. Internet streaming simd extensions. Computer. 1999;32(12):26–34.
36. Waddington D, Harris J. Software challenges for the changing storage landscape. Commun ACM. 2018;61(11):136–45.
37. Xu L, Bruck J. X-code: MDS array codes with optimal encoding. In: IEEE transactions on information theory. vol. 45. IEEE; 1999. pp. 272–276.