Adapting STAR Code for High Performance Non-Volatile Memory 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—Erasure codes have been widely used in various data storage products and systems, from disk arrays to data center systems. With the rapid development of non-volatile memory (NVM) technology, I/O throughput of GBs/sec to 10s of GBs/sec is becoming reality. It thus calls for high performance erasure codes with much faster encoding and decoding speeds to match NVM throughputs. In this paper, we evaluate and adapt computing performance of STAR Code, a class of erasure code that uses only XORs for encoding and decoding to correct up to 3 erasures (disk failures), and compare its performance with traditional Reed-Solomon code currently being used in many storage systems. Extensive experimental measurements show that STAR with Intel’s SSE and parameters such as block size properly adapted for practical NVM disks and systems can outperform Reed Solomon code significantly in both encoding and decoding speed.

I. INTRODUCTION

The emerging Non-volatile Memory (NVM) technologies, such as Phase Change Memory [16] and STT-RAM [8], offer an opportunity for a new generation of data storage devices with low-latency access, high bandwidth, and persistency. Their performance can be much higher than traditional hard drives and NAND based SSDs (Solid State Disks). A single NVM based disk can now reach I/O throughput of GB/s and will reach 10s of GB/s in near future [25] [17]. We can anticipate that a new generation of high performance data storage disk arrays and systems using NVM based disks will soon reach overall I/O throughputs of 10s to 100s GB/s. Other components of such storage systems thus need to be adapted to match high throughput of NVM devices in order to fully utilize performance potential of NVM devices. One such component is erasure code used for data reliability.

Over the years, erasure codes, which provide data reliability in an economical way, are widely employed in various data storage systems, such as Amazon’s S3 [2] [20], Google’s file system [9] and Microsoft’s Azure [11] [15]. Erasure Codes perform extra computations called encoding to generate redundant parity data, when data is written. With the parity data stored, original data can be reconstructed by decoding operations, when device failures occur and data stored on them are lost. By employing a proper erasure code, a storage system can gain the same degree of data reliability as using simple data replication or mirroring of storing multiple copies of the original data, but use storage space and network bandwidth more efficiently. A number of different erasure codes have been introduced, such as the versatile Reed-Solomon Code [24] and various XOR based array codes [4] [26] [7] [12]. Some coding libraries have been developed, such as Jerasure [21] and Intel’s ISA-L [14], and widely used in various storage systems.

In most current data storage systems, as encoding and decoding operations of proper erasure codes, mostly the Reed-Solomon code, have much higher computation bandwidth (in 100s of MB/s to lower GB/s on modern CPUs) than hard disks (10s to lower 100s of MB/s) or even NAND based SSDs (100s of MB/s), encoding and decoding operations of erasure code have not been a performance bottleneck in data storage systems. But with the advent of NVM based storage devices with GB/s to 10s of GB/s throughputs, erasure codes need to be adapted to achieve matching encoding and decoding throughput accordingly. In this paper, we discuss and evaluate some techniques to adapt a class of XOR based array code, the STAR code, to achieve 10s of GB/s encoding and decoding throughputs, thus matching NVM device high performance.

Throughout this paper, NVM devices are only treated as persistent storage block devices, just as NAND based SSDs for most data storage systems and applications, not as a replacement of DRAM based memory for computation. Thus other properties of NVM, such as low power use, random access and erase-then-write, are irrelevant in the following discussions. The only relevant property of NVM disks is their high I/O throughput. Accordingly, encoding and decoding computations are still performed using DRAM based memory.

The paper is organized as follows: Section II introduces some basic background of erasure codes in general; Section III describes preliminary experimental results for performance evaluation; Section IV-A discusses in detail some adaption techniques to significantly improve encoding and decoding performance of the STAR; Section V shows performance improvement of the adapted STAR code by comparison with performance of several open source state-of-art implementations of the Reed-Solomon code; and Section VI draws some conclusions.
II. ERASURE CODES AND RELATED WORK

Erasure codes are mathematical transformations to provide reliability for various data storage systems [19], [23]. 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 [19]. 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.

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, \(n = k + m\) 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 \(\text{blocksize} = \text{packetsize} \times r\). 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.

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 [24]. 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 [26], the RDP code [7], and the STAR code [12] 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].

As will be seen in the following sections, XOR based array codes can achieve much higher encoding and decoding throughputs than finite field operation based Reed-Solomon codes. Thus in this paper, we focus discussions on performance adaption techniques on array codes, using STAR code as an example, since it has the highest data reliability among existing MDS array codes. While some technique details are specific to STAR code, general principles discussed later do apply to other array codes as well, which will be useful to practical data storage system employing these codes.

A. Reed-Solomon code

Reed-Solomon (RS) Code dates back to the 1960s [24]. 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 words to create a codeword, consisting of \(k\) data and \(m\) parity words. This process is illustrated as in Figure 2 where \(k = 4\) and \(m = 2\).

![Fig. 2: Reed-Solomon code encoding process.](image)
deleting rows of \( G^T \), inverting it, and multiplying the inverse by the surviving words. 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) \[18\], multiplication is more complicated, typically implemented with multiplication tables or discrete logarithm tables \[10\]. This makes Reed-Solomon code computationally expensive.

### B. STAR codes

<table>
<thead>
<tr>
<th>Data columns</th>
<th>Parity column III</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>E</td>
</tr>
<tr>
<td>B</td>
<td>A</td>
</tr>
<tr>
<td>C</td>
<td>B</td>
</tr>
<tr>
<td>D</td>
<td>C</td>
</tr>
<tr>
<td>E</td>
<td>D</td>
</tr>
</tbody>
</table>

Fig. 3: STAR code: generating parity column III.

STAR Code is designed and proposed in 2007 \[12\]. 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 \[12\] at this point. For STAR code, \( m = 3 \), while \( k \) is designed to be a prime number for best computation performance. 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 \[12\].

### III. Measurement Setup

In this paper, all throughput measure 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, the gcc option is set as −O3 consistently.

For the same reason as in other tests \[23\] \[13\] \[22\], 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.

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, 2KB respectively, and the testing data size is 1000 stripes (codewords). (Throughout this paper, we use 1000 stripes to measure all tests, which are enough to smooth out measurement fluctuations.) 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, at around 20-40 GB/s, while memcpy is slightly faster than XOR, which is not surprising.

### IV. Adapting STAR Code

In this section, we use STAR Code as an representative for XOR based array codes to explore various adaption techniques to improve encoding and decoding performance. First we discuss a data parallelization technique in software, then show how tuning a coding parameter, namely block size affects encoding and decoding performance; and lastly propose a scheduling technique to improve decoding performance.

To fully invoke the best performance of STAR Code, we first try to integrate Intel’s SSE \[13\] into STAR Code. Then we adapt different coding parameters. As introduced in previous sections, many factors will impact on both RS Code and STAR Code, such as the size of each encoding/decoding strip blocksize, and the total number of data units \( k \). For STAR Code only, a prime number \( p \) needs to be decided in order to process encoding and decoding, thus, this number will influence on the performance as well. Lastly, we try to invovle decoding path into STAR Code, and discuss the possible implementation onto hardware level.

#### A. Data Parallelization by using Intel’s SSE

Parallelization in data to improve performance is a well known technique in computing systems. Over the past decades, Intel’s Streaming SIMD (Single Instruction Multiple Data) Extensions (SSE) has been widely used in various systems for cryptographic operations \[1\], erasure coding \[22\] \[14\] and other multimedia media processing operations \[6\], and now is integrated inside most modern microprocessors of many brands \[15\]. In general, with Intel’s SSE, eight 128-bit general-purpose registers are integrated, addressed from XMM0 to XMM7. Each one of these eight registers is composed of four 32-bit single-precision, floating-point numbers, numbered from 0 to 3. As shown in Figure 5, everytime an logic operation is applied (in this case, XOR), 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 results in the destination operand.

Both Jerasure and ISA-L have seen their RS Code encoding and decoding performance accelerated by integrating Intel’s SSE [22] [14]. Now we examine how Intel’s SSE can accelerate encoding and decoding performance of STAR Code too. In order to achieve this goal, we have implemented two difference versions of STAR Code: one without using Intel’s SSE, and the other exploiting Intel’s SSE. In particular, we mainly adopt the following operations of Intel’s SSE, in order to accelerate 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 the following performance measurement tests, m is set to 3, which means decoding is to recover 3 erasures. k ranges from 6 to 17, and the blocksize is set to 1KB and 2KB, respectively. The results are shown in Figure 6 with x-axis representing k while y-axis representing throughput in GB/sec. Apparently, with Intel’s SSE operations, both encoding and decoding operations achieve much higher throughputs. It is not surprising that the improvement brought by Intel’s SSE is rought between 40% to 80%. Thus first technique a storage practitioner should adopt to enjoy significant performance improvement in encoding and decoding operations for XOR based array codes is to use Intel’s SSE instructions for XORs on 128-bit units instead of smaller ones. This is an easy implementation since Intel’s SSE is almost ubiquitous on all modern CPUs.

B. Adapting p

As we discussed in section I there needs to be a prime number p pre-decided, in order for STAR Code to encode and decode. In this structure, each strip consists of r = p – 1 packets, which is the smallest data unit in STAR Code encoding/decoding structure. The only constraint p needs to satisfy is that it can not be less than k. In previous tests, we set p = 17, so that r = p – 1 will be 16, which will always be a integral factor of whatever blocksize is.

In this section, we long for the impact of p. Firstly, instead of being consistently set as 17, p is altered to be the closest prime number that is larger than or equal to k, e.g. if k = 6 or 7, then p = 7. We compare the testing results of these two different settings. The results are shown in Figure 7 where x-axis represents k from 6-17, while y-axis represents speed in GB/sec. The blocksize is set to 1KB, 2KB and 4KB respectively. When testing encoding, we encode with m = 3, while when testing decoding, we decode with m = 1, 2, 3, separately. It is clear from the results that with p consistently set to 17, encoding/decoding performance of STAR Code is slightly better than that with p not always set to 17, although the difference is not too much. Yet, the curves are more smooth when p is consistent as 17. This is because, with r = 16, the blocksize is always some integral multiple of it. Since we integrate Intel’s SSE inside of STAR, in this way, each one of the strips can be exactly divided and fit onto the registers, as introduced in section IV-A. Thus, makes the XOR operation in a more smoothly way.

The results indicate that, smoother curves can be achieved by blocksize being dividable by r. In order to achieve this goal, p = 17 or 257 is a reasonable choice. 257 is a prime number, and by setting p as 257 constantly, with the same blocksize, the packetsize = blocksize/r is just 1/16 of that when p = 17, e.g., when blocksize = 2KB, packetsize = 8B for p = 257, byt 256B for p = 17. Typically, a larger packetsize can make better use of L-1 and L-2 caches for XORs, and thus encoding/decoding performance. The following experiments are to verify this behavior.

Figure 8 shows the result of encoding/decoding performance of STAR Code with p = 17 vs. p = 257. The x-axis represents k from 6 to 17, while the y-axis represents the speed in GB/sec. The blocksize is set to 1KB, 2KB and 4KB respectively. When testing encoding, we encode with m = 3, while when testing decoding, we decode with m = 1, 2,
3, separately. The results clearly show that STAR Code with \( p = 17 \) constantly outperforms that with \( p = 257 \) by around 10% to 15%, whatever blocksize, \( k \) or \( m \) is. Thus, throughout this paper, we set \( p \) as 17 consistently, for all other evaluations outside this subsection.

C. Impact of Decoding Path

As we discussed in section 1, XOR is the only operation used in STAR Code. Generally speaking, to either encode or decode, STAR will just try to locate a series of packets from one stripe, and perform XOR between them. Within encoding, if \( p \) is given, the location of this series of packets will be automatically decided accordingly. Similarly, in decoding process, if \( p \) and the erasure indexes are given, the location of the packets to be computed can also be decided by calculation. What’s more, every stripe will share the exact same location from this calculation. And this calculated locations of the packets needed for decoding, is called Decoding Path.

In this section, we examine the impact of bringing decoding path into STAR Code decoding process. As before, we implement two different versions of STAR Code, in one of which, the indexes of packets needed for decoding are calculated within each individual stripe; while in the other version, the decoding path is only calculated within the first stripe, and then is stored in cache, so that from the second stripe on, no more computation of decoding path is needed. Instead, STAR can directly reads the path, gets the indexes of the needed packets and simply performs XOR operation between them.

The results are shown in Figure 9 where x-axis represents blocksize from 16B to 4KB, while y-axis represents speed in GB/sec. The \( k \) is set to 10 consistently. The results show the decoding performance is improved by 10% to 50% roughly, which is no surprising. This improvement is more obvious when decoding 3 erasures, this is because, when there are 3 erasures, the decoding path is more random and unpredictable. It takes way more computation to calculate (rather than decoding 2 erasures). With decoding path computed and stored after the first stripe, all the following stripes can directly locate the indexes of packets they need. When STAR Code is decoding, the total number of stripes can be large (in this case, 1000). So this can save a big amount of index computations. Thus, improves the decoding speed. Throughout this paper, we use the version of STAR, with decoding path implemented, except this subsection.

D. Hardware Implementation

With all being discussed and demonstrated in this section, it is clear that STAR Code can be invoked to its best performance, by correctly adapting Intel’s SSE, \( p \), and decoding path. Obviously, this process could be achieved easily by giving the adequate system settings when configuring, or with corresponding usage of applications on the software level. However, this could be also achieved, by some adaptations on the hardware level.
Fig. 8: Encoding and Decoding performance of STAR for $p = 17$ vs. $p = 257$ with blocksize = 1KB, 2KB, 4KB, respectively.

Fig. 9: Adapting Decoding Pattern into STAR Code

Intel’s SSE has already been integrated onto most of the microprocessors. And to make the most use of this technology, the memory units on NVM system could be organized in array of registers, one of which contains 16 units of data. In this way, when reading the data from the memory, the data could be matched with the registers that SSE provides on microprocessors. Thus, in return, invokes better speed.

Besides, there could be an extra piece of small memory unit integrated on the chip, used for storing the decoding path. With this, the decoding path no longer needs to occupy the precious space in cache, while when decoding, the decoding path can still be easily directed and used.

V. PERFORMANCE EVALUATION

After IV, we have invoked the best performance out of STAR Code, with certain adaptions. In this section, we compare the adapted STAR Code with RS Code. We use Jerasure and ISA-L to represent the RS-Code. Many factors will greatly influence the practical performance of both STAR Code and RS Code, namely, the number of data units $k$, the blocksize, or the cache size of the testing machine. Thus, this section can only provide a general guidance, rather than some accurate prediction. All experiments are conducted and measured with 1000 stripes (codewords) to make full use of L-1 to L-3 caches (as mentioned in III), and the results are averaged. Note that for STAR Code, $p$ is set to 17 constantly, and for RS Code, $w$ is set to 8.

Both STAR Code and RS Code are integrated with Intel’s SSE.

A. Comparison on blocksize

Among all factors related to STAR Code and RS Code, blocksize is one of the most practical ones. It is easy to be changed and since it is the smallest unit in the structure, it affects the general performance greatly. In this subsection, we long for the comparison between STAR Code and RS Code, with ranging blocksizes from 1KB to 32KB. $k$ is set to 10 for both STAR Code and RS Code, which is just a reflection of usual application or system configuration. When testing encoding, $m$ is set to 3, while, when testing decoding, $m$ is set to 1, 2, 3, respectively.

The results are shown in Figure 10, where the x-axis represents blocksize, while y-axis represents speed in...
GB/sec. As is indicated from the results, speed of both STAR and Jerasure increases with larger blocksize. This is due to better cache uses for either finite field multiplication or XOR operations. On the other hand, ISA-L performs stably, which could possibly be caused by the implementation of assembly language and being already optimized using different levels of caches, particularly the L-1 cache. Regardless, with adequate adaptations, STAR Code outperforms both Jeasure and ISA-L, especially with blocksize $> 4$ KB.

![Performance comparison between STAR Code and RS Code for different blocksize](image)

**Fig. 10:** Performance comparison between STAR Code and RS Code for different blocksize.

**B. Comparision on k**

In this subsection, we are to compare STAR Code with RS Code, on different $k$, from 6 to 17. Based on results from [V-A], we set blocksize to be 4KB, where both STAR Code and RS Code can achieve a relatively desirable speed. When testing encoding, $m$ is set to 3, while, when testing decoding, $m$ is set to 1, 2, 3, respectively. The results are shown in Figure [11]. Consistent to section [V-A], STAR Code performs the best among the three tested libraries. Particularly, when there is only one erasure ($m = 1$), which is likely to be the most common case in real-life systems, STAR significantly beats RS Code by roughly 150%, reaching a decoding speed of around 25 GB/sec, which is very close to the baseline speed discussed in section [4].

![Performance comparison between STAR Code and RS Code for different $k$](image)

**Fig. 11:** Performance comparison between STAR Code and RS Code for different $k$.

**VI. Conclusions**

.....

**REFERENCES**