EC-Bench: Benchmarking Onload and Offload Erasure Coders on Modern Hardware Architectures *

Haiyang Shi, Xiaoyi Lu, and Dhabaleswar K. (DK) Panda

Department of Computer Science and Engineering, The Ohio State University
{shi.876, lu.932, panda.2}@osu.edu

Abstract. Various Erasure Coding (EC) schemes based on hardware accelerations have been proposed in the community to leverage the advanced compute capabilities on modern data centers, such as Intel ISA-L Onload EC coders and Mellanox InfiniBand Offload EC coders. These EC coders can play a vital role in designing next-generation distributed storage systems. Unfortunately, there does not exist a unified and easy way for distributed storage systems researchers and designers to benchmark, measure, and characterize the performance of these different EC coders. In this context, we propose a unified benchmark suite, called EC-Bench, to help the users to benchmark both onload and offload EC coders on modern hardware architectures. EC-Bench provides both encoding and decoding benchmarks with tunable parameter support. A rich set of metrics, including latency, actual and normalized throughput, CPU utilization, and cache pressure, can be reported through EC-Bench. Evaluations with EC-Bench demonstrate that hardware-optimized offload coders (e.g. Mellanox-EC) have lower demands on CPU and cache compared to onload coders, and highly optimized onload coders (e.g., Intel ISA-L) outperform offload coders for most configurations.

1 Introduction

Replication, a redundancy scheme that replicates data across multiple machines and racks, is widely used to guarantee high reliability and availability against the most failure scenarios in distributed storage systems. Since the data being generated increases rapidly every day, petabytes of storage in today's data centers are becoming common. As a result, distributed systems cannot tolerate such a significant storage overhead brought by using N-way replication, even though disk storage is inexpensive today.

To this end, latest distributed storage systems, such as Google Colossus [6], Facebook HDFS-RAID [1, 36], the Quantcast File System [27] and Microsoft Azure Storage System [15], are transforming to the use of Erasure Coding (EC) scheme, which offers high reliability and availability at a prominently low storage overhead [42, 35]. For instance, Reed-Solomon [34] is a popular family of erasure codes used in Google Colossus, Facebook HDFS-RAID, and many others. The Reed-Solomon codes with a $6+3$ configuration, i.e., three parity chunks for every six data chunks, which delivers the same level of fault tolerance as 4-way replication scheme does, has a storage overhead of 50%, while 4-way replication has a storage overhead of 300%.

* This research is supported in part by National Science Foundation grants CCF#1822987, CNS#1513120, IIS#1636846, and OAC#1664137.

The trade-off of deploying erasure codes in distributed storage systems instead of replication is performance. The use of erasure coding results in a significant increase in computation overhead due to the time-consuming EC encoding and decoding operations. With such a trade-off, the erasure-encoded distributed storage systems should benefit from modern high-performance hardware architectures. The advancements in CPU/GPU architectures and network interconnects have enabled the design of high-performance erasure coding libraries [16, 24] for alleviating the compute overheads involved in erasure coding-based storage resilience. This motivates us to believe that erasure coding could be a viable primary fault-tolerance mechanism for next-generation distributed storage systems.

High-performance EC coders can be categorized in two general ways: (1) EC Onload, where host-based libraries such as Jerasure [30] and Intel ISA-L [16] are employed, and, (2) EC Offload, wherein Mellanox InfiniBand HCA and GPU-like accelerators based libraries such as Gibraltar [8] and Mellanox-EC [23] are leveraged. With the increased compute and remote I/O required for computing and distributing EC-coded files, EC onload can enable higher storage efficiency that is inherent to erasure-coded storage, at the cost of performance and CPU usage. On the other hand, the high CPU usage can be alleviated with the help of EC Offload designs, that offload computation to the Mellanox HCAs or GPU devices, but suffer the loss of performance due to its limited compute capabilities in comparison to CPU cores.

As we can see, efficient EC coders can play a significant role in designing next-generation distributed storage systems. However, each of these EC coders has different APIs, implementations, and performance characteristics. To guide the users to choose an appropriate one for their target platforms, the community needs a unified and easy-to-use benchmark suite to measure the performance and expose the insights of different coders.

Unfortunately, there does not exist such a benchmark suite for distributed storage systems researchers and designers to benchmark, measure, and characterize the performance of these different EC coders. To address this issue, in this paper, we propose a unified benchmark suite, called **EC-Bench**, to help the erasure coding researchers and distributed storage system designers to benchmark both onload and offload EC coders on modern hardware architectures. EC-Bench provides both encoding and decoding benchmarks with tunable parameter support. The supported parameters include the number of data chunks, the number of parity chunks, the number of bits in a word, and the size of each chunk. To help the users to understand the EC coders in multiple dimensions, EC-Bench reports a rich set of metrics including latency, actual and normalized throughput, CPU utilization, and cache pressure.

With EC-Bench, we conduct experiments on four open-source libraries (i.e., Jerasure [30], ISA-L [16], Gibraltar [8], and Mellanox-EC [23]) to evaluate the performance of onload and offload erasure coders. Our in-depth evaluation exposes impressive performance insights for different coders on modern CPU, GPU, and InfiniBand architectures. For instance, the experiments illustrate that hardware-optimized offload coders (e.g., Mellanox-EC) have less CPU utilization and cache pressure than onload coders, and highly optimized onload coders (e.g., Intel ISA-L) perform much better than offload coders due to the use of advanced instruction sets.

The rest of the paper is organized as follows. Section 2 presents the necessary background on EC. Section 3 presents our proposed design for EC-Bench. Section 4 describes our detailed evaluation. Section 5 discusses related studies. Finally, we conclude in Section 6.

2 Background

2.1 Erasure Coding

Conventionally, a storage system tolerates faults by replicating data to different nodes and racks. For example, GFS, HDFS, and Ceph apply 3-way replication as their default storage mechanisms [11, 39, 43]. Unfortunately, replicating a tremendous amount of data can incur significant storage overhead. Therefore, erasure coding (EC), which can offer the same reliability as or higher than replication with much lower storage overhead, becomes an attractive alternative. The Reed-Solomon (RS) code and its variations are the most popular erasure codes employed in distributed file systems (e.g., HDFS, Ceph, QFS, Google Colossus, Facebook f4, Baidu Atlas and Backblaze [26, 18, 6, 3, 27, 5, 43, 4]). In general, the input data is split into chunks with a fixed size (i.e., *chunk_size*). An RS coder, denoted as $RS(k,m)$, computes m parity chunks for k data chunks. These $k+m$ chunks are organized as a group called *stripe*. For chunks belonging into the same stripe, $RS(k,m)$ is able to recover the entire stripe from up to m chunk losses, with a storage overhead of m/k. In contrast, the replication scheme has to store $m+1$ replicas to achieve the same reliability; thus the storage overhead of replication is as high as m. For example, the $RS(6,3)$ code has a storage overhead of 50% and delivers the same fault-tolerance as 4-way replication that incurs a 3x overhead. One of the disadvantages in applying erasure coding to storage systems, however, is high pressures of erasure operations on system performance.

2.2 Onload and Offload Erasure Coders

To overcome the high computational costs involved with erasure coding, two broad categories of coders have been proposed in the community to take advantage of modern hardware capabilities: (1) onload coders, which are highly optimized for advanced CPU capabilities (e.g., Intel SSE [41] and AVX [17]), and, (2) offload coders, which offload erasure operations to accelerators (e.g, GPU [8], Host Channel Adapters (HCA) [24]). These hardware-optimized erasure coders can potentially facilitate EC to be employed as a viable choice for fault-tolerance in modern distributed storage systems.

3 EC-Bench Design

In this section, we discuss the design details, parameter space, and main metrics of our benchmarking framework, i.e., EC-Bench.

3.1 Design

EC-Bench consists of two benchmarks, one for encoding and one for decoding.

Encoding Benchmark: For encoding benchmark, a large in-memory data file of size D are split into multiple data blocks of size $k \times$ *chunk_size*. Each encoding operation of the evaluated erasure coder encodes a piece of data block into $k+m$ data and parity

chunks.

Decoding Benchmark: In order to generate data and parity chunks (i.e., stripes), such that we can mimic chunk corruption by nullifying some chunks, a preprocessing stage before performing decoding operations is necessary. In the preprocessing stage, it encodes a large in-memory data file of size D into multiple encoded stripes of size $(k+m) \times chunk\_size$, and randomly zeros m chunks out of $k+m$ chunks in each encoded stripe. After the preprocessing stage, each decoding operation of the evaluated erasure coder recovers an encoded stripe. To fairly compare all erasure coders, both data and parity chunks need to be recovered in the benchmark. For some erasure coders, such as Gibraltar [8], which only recover data chunks in decoding, we will re-encode to recover corrupt parity chunks.

3.2 Parameter Space

As aforementioned in Section 2, the most important parameters for all erasure coders are the number of data chunks k, the number of parity chunks m, the number of bits in a word w, and the size of each chunk *chunk_size*. Therefore, in EC-Bench, the values of k, m, w, and *chunk_size* may be chosen at the discretion of the user and according to the constraints of erasure coders to evaluate.

3.3 Metrics

In addition to latency and throughout, which are the most typical metrics for benchmarking erasure coders, we also introduce *CPU utilization* and *cache pressure* as main metrics to evaluate onload and offload erasure coders. In this section, we clarify the definition and describe the approach used for each metric in EC-Bench.

Latency The latency in EC-Bench is defined as the time spent on erasure coding operations (i.e., encoding and decoding).

Throughput The throughput in EC-Bench is defined as the size of data and parity chunks divided by the time spent on erasure coding operations (i.e., encoding and decoding). Let D denote the size of k data chunks, and t denote the time consumed by erasure coding operations. Such that the size of each chunk is D/k. As illustrated in Section 3.1, for both encoding and decoding operations, the erasure coder operates on k chunks and generates another m different chunks. It means that each benchmark will output an in-memory data file of size $D \cdot (k+m)/k$ given an input of size D. Hence, the definition of throughput turns out to be:

$$\text{Thr} = \frac{D}{t} \cdot \frac{k+m}{k} \tag{1}$$

As shown in the equation, the value of throughput is related to k and m. Sometimes, however, it is helpful to compare the throughput across different combinations of k and m. Therefore, we also introduce the normalized throughput as a metric. Since for both encoding and decoding operations, it generates $D \cdot m/k$ bytes worth of coding data. Studies [29, 44] demonstrate that it takes $k-1$ XOR operations to produce a byte.

Therefore, if we define the normalized throughput as the number of XOR operations taken place in erasure coding operations divided by the time consumed, the metric is fair for all combinations of k and m. Thus, the normalized throughput is represented as:

$$\text{Thr}_{\text{norm}} = \frac{D}{t} \cdot \frac{(k-1)\cdot m}{k} = \frac{(k-1)\cdot m}{k+m} \cdot \text{Thr} \tag{2}$$

CPU Utilization A well-known advantage of offload architecture is the low-consumption of CPU cycles, which frees up CPU for computation tasks and finally increases overall system efficiency and performance. Therefore, another important metric to differentiate onload and offload erasure coders in EC-Bench is CPU utilization. To precisely get the CPU utilization of each evaluated erasure coder, we employ PAPI [40] APIs to collect the total number of CPU cycles consumed by erasure coding operations. We define CPU utilization as the total number of consumed CPU cycles divide by the time spent on erasure coding operations. Its equation representation is:

$$\text{CPU Utilization} = \frac{\text{CPU cycles}}{t} \tag{3}$$

Cache Pressure Concerning the architecture characteristics of onload and offload erasure coders, cache pressure is another vital metric introduced in EC-Bench. With the fact that, for a specific erasure coder, the maximum performance point is achieved when the coder makes the best use of the L1 cache [29], cache pressure is at least a complementary to other metrics to explore performance differences between onload and offload coders. For who is developing a new erasure code, cache pressure may as well be a non-trivial metric to analyze performance bottleneck. PAPI APIs are used to collect the number of cache misses in different cache levels. Therefore, cache pressure is defined as the total number of L1 cache misses divided by the time spent on erasure coding operations. Therefore, its formula is:

$$\text{Cache Pressure} = \frac{\text{L1 Cache Misses}}{t} \tag{4}$$

4 Evaluation

In this section, we conduct experiments on four open-source libraries with EC-Bench to evaluate the performance of onload and offload erasure coders. This section also includes additional details on our experimental setup and results.

4.1 Open Source Libraries

These four erasure coder libraries are freely available from various resources on the Internet. The following list represents their descriptions.

Jerasure: Jerasure [30] is a CPU-based library released in 2007 that supports a wide variety of erasure codes. The w of Reed-Solomon code in Jerasure could be 8, 16, or

32.
ISA-L: Intel Intelligent Storage Acceleration Library (ISA-L) [16] is a collection of optimized low-level functions including erasure coding. The erasure coding functions are optimized for Intel instructions, such as Intel SSE [41], vector [17], and encryption instructions. The w of Reed-Solomon code in ISA-L is fixed to 8.
Gibraltar: Gibraltar [8] is a GPU-based library for Reed-Solomon coding. The Reed-Solomon code in Gibraltar is based on $GF(2^8)$, which means it has a fixed $w = 8$.
Mellanox-EC: Mellanox-EC [24] proposed by Mellanox is an HCA-based library for Reed-Solomon coding. The erasure coding operations are handled in host channel adapters (HCA). The w of Reed-Solomon code could be 4 and 8 in the latest ConnectX-5 IB NICs.

4.2 Experimental Setup

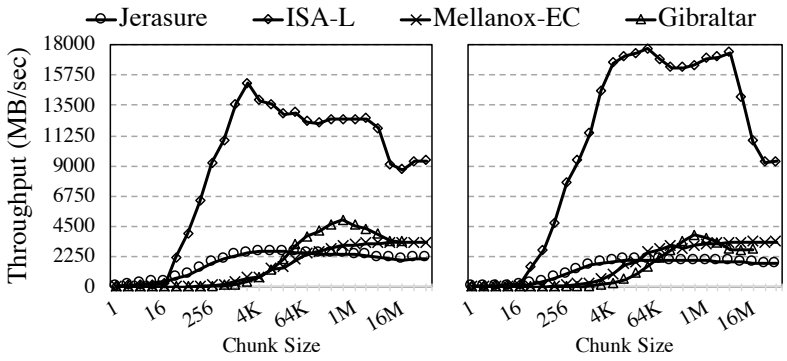
Our cluster consists of 20 nodes, and each is equipped with 2.40GHz Intel(R) Xeon(R) CPU E5-2680 v4 (28 cores, 32KB L1 cache, 256KB L2 cache, and 35MB L3 cache), 128GB DRAM, two K80 GPUs, and a ConnectX-5 IB-EDR (100 Gbps) NIC. The operating system employed in the cluster is CentOS 7.2. Other necessary drivers and libraries are CUDA 8.0, Mellanox OFED 4.2, PAPI 5.2.0.0 with perf 3.10.0, Jerasure 2.0, ISA-L 2.18.0, Gibraltar [1], and Mellanox-EC [2]. Note that Jerasure in our experiments is compiled without SSE support, such that Jerasure represents onload erasure coder with common instruction sets while ISA-L with advanced instruction sets.

Experiments in this paper are all conducted with Reed-Solomon code as it is the only common erasure code among chosen libraries as illustrated in Section 4.1. We also fix the value of w into 8, such that all onload and offload coders are comparable. Let $RS(k,m)$ denote the configuration of Reed-Solomon code computing m parity chunks for k data chunks. We examine onload and offload coders with four popular configurations, $RS(3,2)$, $RS(6,3)$, $RS(10,4)$ and $RS(17,3)$ used by HDFS, Ceph, QFS, Google, Facebook, Baidu and Backblaze [26, 18, 6, 3, 27, 5, 4], etc.

4.3 Experimental Results

It is well-known that decoding operations are similar to encoding operations for RS code. The throughput performance of encoding and decoding for $RS(3,2)$ depicted in Figure 1 demonstrates that encoding performance and decoding performance of all selected coders have similar trends. One interesting observation in the figure is that the decoding performance of ISA-L to recover m (m equals 2 in Figure 1.) lost chunks is better than its encoding performance to generate m parity chunks. In the experiment, the m-by-m matrix used for decoding has a smaller size than the generator matrix (i.e., a k-by-m matrix) for encoding, such that the decoding operation requires less compute power; thus, erasure coders, especially high-performance erasure coders such as ISA-L, deliver better decoding performance.

[1] Github: `https://github.com/jaredjennings/libgibraltar`,
commit: c93f9d8c3be70ded173822cdca2e51900a3f5ed1
[2] Github: `https://github.com/Mellanox/EC`,
commit: 00bf091aa14322baf4425f8a6d5d134e91fe2a5c



(a) Encode (b) Decode

Fig. 1: Throughput Performance with Varied Chunk Sizes for RS(3, 2)

Since both encoding performance and decoding performance of different coders have similar tendencies, we only show encoding results in this section due to space limitation. In our experiments, Gibraltar coder is not able to run with $chunk\_size = \{32MB, 64MB\}$, such that corresponding numbers in the following figures are left blank.

Throughput Figures 2-5 depict throughput performance comparisons among onload and offload coders with various chunk sizes ranging from 1 byte to 64 MB. Normalized throughput is showing on the right-hand-side y-axis, and each data point in the figures corresponds to two values, i.e., throughput and normalized throughput. In all experiments, onload coders outperform offload coders for small chunk sizes. For instance, in Figure 2, both Jerasure and ISA-L perform better than Mellanox-EC and Gibraltar with chunk sizes smaller than $32KB$. On the other hand, throughput performance of offload coders improves significantly with increasing chunk sizes, and offload coders are able to defeat some onload coders if chunk size is large enough. Figure 2 demonstrates that Mellanox-EC and Gibraltar coders outperform Jerasure once chunk size is larger than $32KB$. The reason behind the increasing throughput of offload coders with growing chunk sizes is that large chunk sizes alleviate the overhead of transferring data from host to device [10]. In Figures 3-5, we observe trends similar to the trend demonstrated in Figure 2.

Normalized Throughput After being normalized, throughput performance across different configurations is comparable [29]. Figure 6 shows how the normalized throughput performance of onload and offload coders changes across multiple configurations (e.g., $RS(3,2)$). ISA-L, Mellanox-EC and Gibraltar coders are sensitive to configuration changes while Jerasure keeps consistent across different configurations. One possible reason behind this observation is that different coders have nonequivalent optimal parallelism supports. ISA-L, Mellanox-EC, and Gibraltar have good support to larger-scale parallel configurations; thus, they perform good with $RS(10,4)$ and $RS(17,3)$. In contrast, Jerasure (compiled without SSE support) prefers smaller-scale parallel configurations; therefore, it achieves its best performance with configuration $RS(3,2)$.

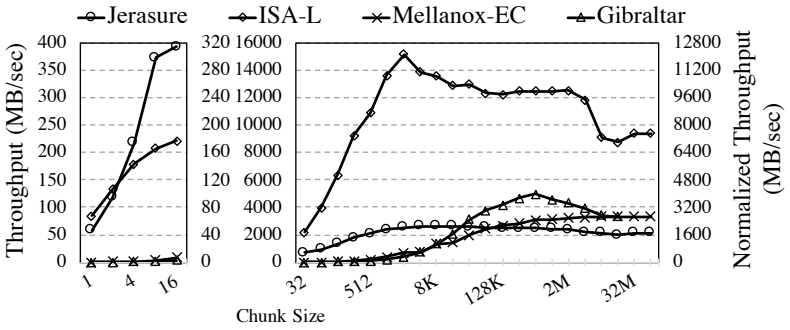


Fig. 2: Throughput Performance with Varied Chunk Sizes for RS(3, 2)

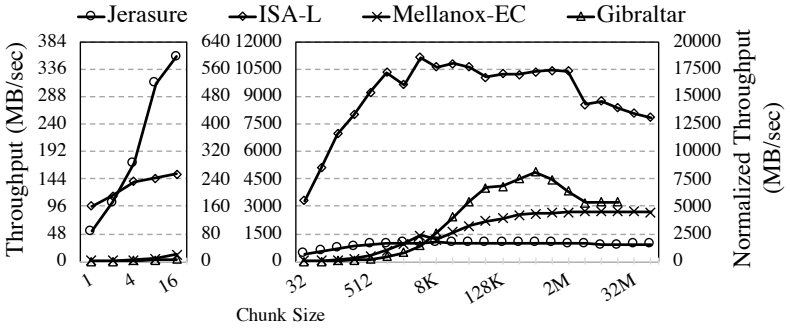


Fig. 3: Throughput Performance with Varied Chunk Sizes for RS(6, 3)

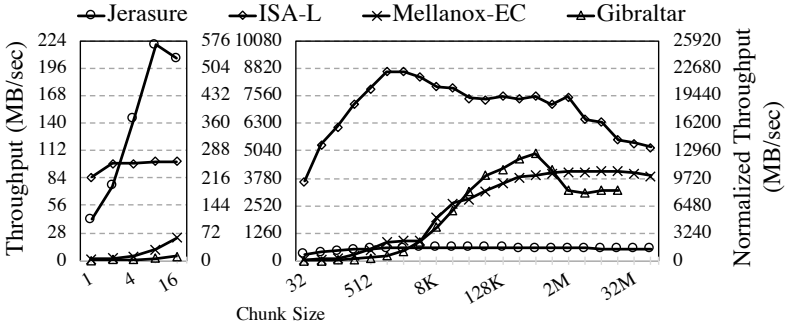


Fig. 4: Throughput Performance with Varied Chunk Sizes for RS(10, 4)

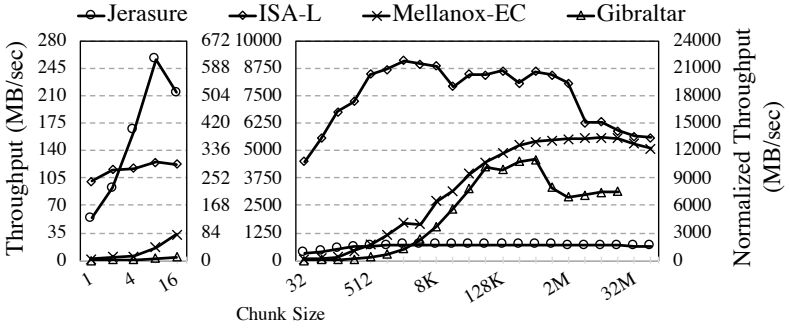


Fig. 5: Throughput Performance with Varied Chunk Sizes for RS(17, 3)

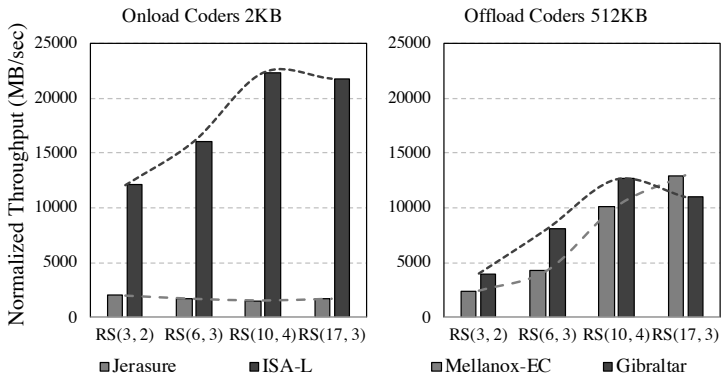


Fig. 6: Normalized Throughput Performance of Onload and Offload Coders across Multiple Configurations. The chunk sizes for onload and offload coders are fixed into one of their near-optimal chunk sizes. In this case, 2*KB* for onload coders and 512*KB* for offload coders.

CPU Utilization Considering CPU utilization of onload and offload coders, Figures 7-10 illustrate that offload coders make better use of CPU cycles to carry out erasure operations compared with onload coders. For example, it shows that, in Figure 7, Mellanox-EC consumes 0.41 million cycles per second while running with a chunk size of 64*MB*. In the meantime, Jerasure and ISA-L take 2950.5 and 2932.23 million cycles per second, respectively. Another observation is that CPU utilization for both onload and offload coders decrease with an increase in chunk size.

Figures 7-10 also show an interesting fact that ISA-L deals with chunk sizes smaller than 32 bytes and other chunk sizes in two different approaches (details in the implementation of function *ec_encode_data_avx2* [2]). That's why in Figures 7-14, there are big jumps in the curves of ISA-L in the cases of 32 bytes.

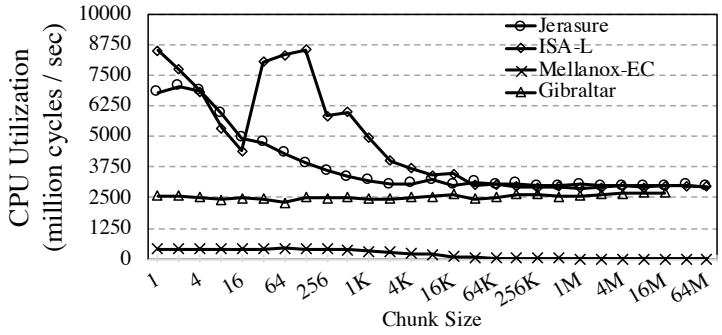


Fig. 7: CPU Utilization with Varied Chunk Sizes for RS(3, 2)

Cache Pressure The cache pressures of onload and offload coders are depicted in Figures 11-14. Though Gibraltar has more L1 cache misses than ISA-L for some chunk sizes, the overall cache pressure introduced by offload erasure coders is less than that

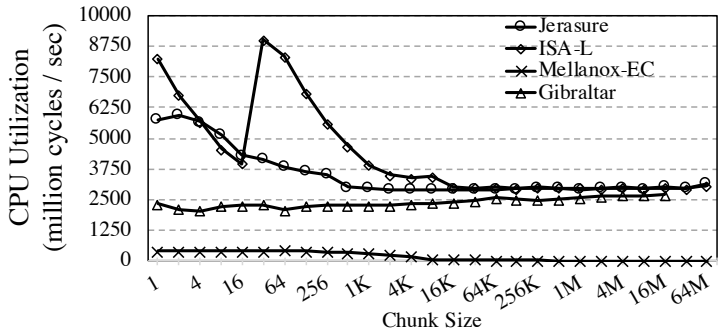


Fig. 8: CPU Utilization with Varied Chunk Sizes for RS(6, 3)

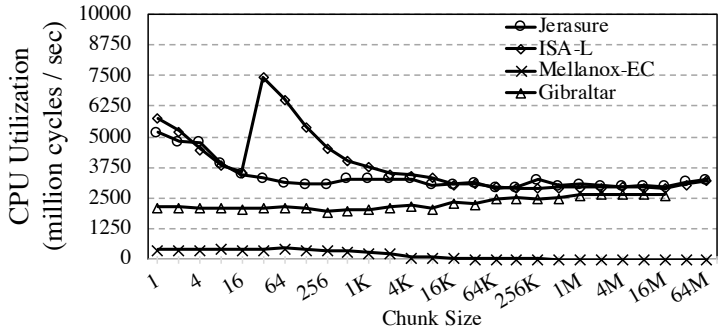


Fig. 9: CPU Utilization with Varied Chunk Sizes for RS(10, 4)

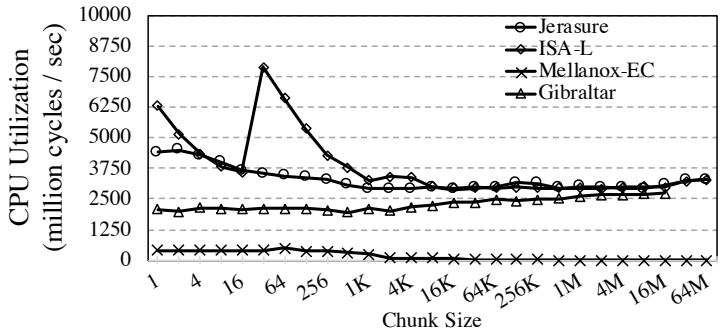


Fig. 10: CPU Utilization with Varied Chunk Sizes for RS(17, 3)

introduced by onload coders. Within all coders, Mellanox-EC influences cache least, while Jerasure has constant pressure on cache for relatively large chunk sizes. The different cache behaviors of ISA-L around $chunk\_size = 32$ across four chosen configurations (e.g., $RS(3,2)$) also indicate the same observation in Section 4.3 that ISA-L has two internal approaches to carry out chunk sizes smaller than 32 bytes and other chunk sizes.

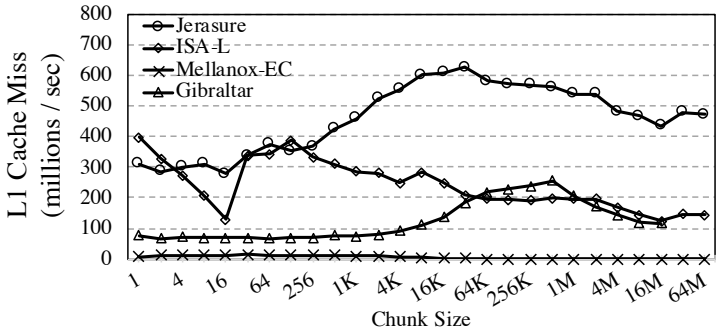


Fig. 11: Cache Pressure with Varied Chunk Sizes for RS(3, 2)

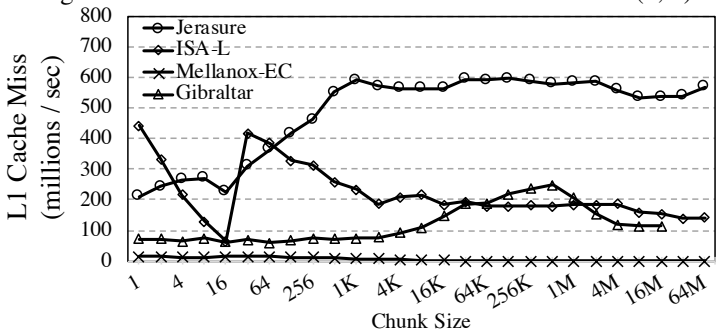


Fig. 12: Cache Pressure with Varied Chunk Sizes for RS(6, 3)

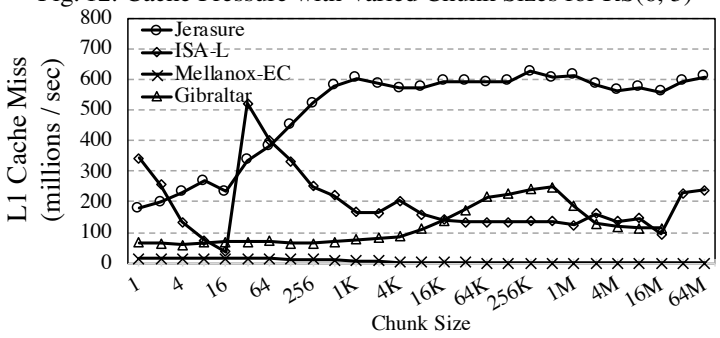


Fig. 13: Cache Pressure with Varied Chunk Sizes for RS(10, 4)

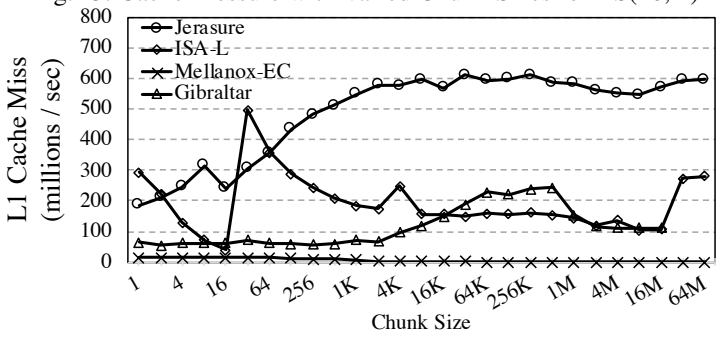


Fig. 14: Cache Pressure with Varied Chunk Sizes for RS(17, 3)

5 Related Work

Over the years, as erasure coding becomes an attractive alternative to replication, several works have been focusing on employing erasure coding for performing data recovery on data centers and benchmarking erasure coders for performance evaluation.

Erasure Coding for Storage Systems: Erasure codes, especially Reed-Solomon code and its variations, have been adopted in famous storage systems [26, 18, 6, 3, 27, 5, 43, 4], because of its higher reliability with lower storage overhead. To further reduce the overhead introduced by erasure coding, some research works are proposed, such as Partial-Parallel-Repair [25], Repair Pipelining [20], and [9, 32, 33]. Several researchers have also designed many other classes of erasure codes to reduce the computational complexity involved in Reed-Solomon codes [7, 14, 19, 13, 15, 12]. In the meantime, erasure coding is also being utilized to design key-value stores, including Cocytus [45], EC-Cache [31], and RDMA-accelerated Memcached with online EC support [37].

Hardware Acceleration and Optimizations for Erasure Coding: Motivated by the advanced features supported by modern CPU architectures, many research works [16, 22, 28] are enabling the design of high-speed EC by taking advantage of instruction sets like SSE, AVX, etc. Along similar lines, [24] and [8] proposed offload approaches to reduce CPU consumption and leverage the capabilities of GPUs and next-generation network adapters, respectively. On the other hand, our previous work [38] has proposed a new concept **Multi-Rail EC**, which enables upper-layer applications to leverage available high-performance hardware in parallel to accelerate erasure coding.

Benchmarking Erasure Coding Libraries: Recent studies have evaluated onload erasure coders with metrics throughput or latency. For instance, [21] performs several experiments to test the running times of some popular software-based onload erasure coders. [29] conducts a throughput performance evaluation and examination of open-source erasure coding libraries and contributes a way to normalize throughput performance across different configurations (e.g., $RS(3,2)$, $RS(6,3)$ and $RS(10,4)$).

The increased focus on employing EC in storage systems and enabling EC on modern hardware serves as a motivation of this paper. Based on our knowledge of modern hardware architectures, we propose a benchmark which supports latency & throughput metrics as well as architecture-related metrics, such as CPU utilization and cache pressure, to fully evaluate different erasure coders, especially onload and offload coders.

6 Conclusion

In this work, we design a benchmark framework (i.e., EC-Bench) for evaluating erasure coders, especially for onload and offload coders. EC-Bench supports four main metrics (i.e., latency, throughput, CPU utilization, and cache pressure), which we think are sufficient to explore the performance characteristics of onload and offload coders fully. Through in-depth performance evaluations of four erasure coders, we demonstrate that EC-Bench is able to reveal their performance differences in terms of throughput, CPU utilization, and cache pressure. The performance results illustrate that onload coders consumes more CPU and cache resources than offload coders (e.g., Mellanox-EC), and highly optimized onload coders (e.g., Intel ISA-L) typically outperform offload coders.

References

1. Facebook's Erasure Coded Hadoop Distributed File System (HDFS-RAID). `https://github.com/facebookarchive/hadoop-20`, 2010.
2. ec_highlevel_func.c. `https://github.com/intel/isa-l/blob/master/erasure_code/ec_highlevel_func.c#L98`, 2018.
3. Apache Hadoop 3.0.0-alpha2. `http://hadoop.apache.org/docs/r3.0.0-alpha2/`, 2017.
4. Backblaze Online Backup. `https://www.backblaze.com/blog/reed-solomon/`, 2015.
5. Ceph Erasure Coding. `http://docs.ceph.com/docs/master/rados/operations/erasure-code/`, 2016.
6. Colossus: Successor to the Google File System (GFS). `https://www.systutorials.com/3202/colossus-successor-to-google-file-system-gfs/`, 2012.
7. Peter Corbett, Bob English, Atul Goel, Tomislav Grcanac, Steven Kleiman, James Leong, and Sunitha Sankar. Row-diagonal Parity for Double Disk Failure Correction. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, pages 1–14. USENIX Association Berkeley, CA, USA, 2004.
8. Matthew Curry, Anthony Skjellum, H Lee Ward, and Ron Brightwell. Gibraltar: A Reed-Solomon Coding Library for Storage Applications on Programmable Graphics Processors. In *Concurrency and Computation: Practice and Experience*, volume 23, pages 2477–2495, 12 2011.
9. Alexandros G Dimakis, P Brighten Godfrey, Yunnan Wu, Martin J Wainwright, and Kannan Ramchandran. Network Coding for Distributed Storage Systems. *IEEE transactions on information theory*, 56(9):4539–4551, 2010.
10. Yusuke Fujii, Takuya Azumi, Nobuhiko Nishio, Shinpei Kato, and Masato Edahiro. Data Transfer Matters for GPU Computing. In *Parallel and Distributed Systems (ICPADS), 2013 International Conference on*, pages 275–282. IEEE, 2013.
11. Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *ACM SIGOPS operating systems review*, volume 37, pages 29–43. ACM, 2003.
12. Kevin M Greenan, Xiaozhou Li, and Jay J Wylie. Flat XOR-based Erasure Codes in Storage Systems: Constructions, Efficient Recovery, and Tradeoffs. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1–14. IEEE, 2010.
13. James Lee Hafner. WEAVER Codes: Highly Fault Tolerant Erasure Codes for Storage Systems. In *Proceedings of the 4th Conference on USENIX Conference on File and Storage Technologies - Volume 4*, FAST'05, pages 16–16, Berkeley, CA, USA, 2005. USENIX Association.
14. C. Huang and L. Xu. STAR : An Efficient Coding Scheme for Correcting Triple Storage Node Failures. *IEEE Transactions on Computers*, 57(7):889–901, July 2008.
15. Cheng Huang, Huseyin Simitci, Yikang Xu, Aaron Ogus, Brad Calder, Parikshit Gopalan, Jin Li, Sergey Yekhanin, et al. Erasure Coding in Windows Azure Storage. In *Usenix Annual Technical Conference*, pages 15–26. Boston, MA, 2012.
16. Intel Intelligent Storage Acceleration Library (Intel ISA-L). `https://software.intel.com/en-us/storage/ISA-L`, 2016.
17. Introduction to Intel® Advanced Vector Extensions. `https://software.intel.com/en-us/articles/introduction-to-intel-advanced-vector-extensions`.
18. Chunbo Lai, Song Jiang, Liqiong Yang, Shiding Lin, Guangyu Sun, Zhenyu Hou, Can Cui, and Jason Cong. Atlas: Baidu's Key-value Storage System for Cloud Data. In *Mass Storage Systems and Technologies (MSST), 2015 31st Symposium on*, pages 1–14. IEEE, 2015.
19. Mingqiang Li and Patrick P. C. Lee. STAIR Codes: A General Family of Erasure Codes for Tolerating Device and Sector Failures. *Trans. Storage*, 10(4):14:1–14:30, October 2014.

20. Runhui Li, Xiaolu Li, Patrick PC Lee, and Qun Huang. Repair Pipelining for Erasure-coded Storage. In *Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC'17)*, pages 567–579, 2017.
21. Michael Luby. Benchmark Comparisons of Erasure Codes, 2002.
22. Aleksei Marov and Andrey Fedorov. Optimization of RAID Erasure Coding Algorithms for Intel Xeon Phi. In *Networking, Architecture and Storage (NAS), 2016 IEEE International Conference on*, pages 1–4. IEEE, 2016.
23. Mellanox. HDFS Erasure Coding Offload Plugin. `https://github.com/Mellanox/EC/tree/master/HDFS`, 2016.
24. Mellanox. Understanding Erasure Coding Offload. `https://community.mellanox.com/docs/DOC-2414`, 2016.
25. Subrata Mitra, Rajesh Panta, Moo-Ryong Ra, and Saurabh Bagchi. Partial-Parallel-Repair (PPR): A Distributed Technique for Repairing Erasure Coded Storage. In *Proceedings of the Eleventh European Conference on Computer Systems*, page 30. ACM, 2016.
26. Subramanian Muralidhar, Wyatt Lloyd, Sabyasachi Roy, Cory Hill, Ernest Lin, Weiwen Liu, Satadru Pan, Shiva Shankar, Viswanath Sivakumar, Linpeng Tang, et al. f4: Facebook's Warm BLOB Storage System. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, pages 383–398. USENIX Association, 2014.
27. Michael Ovsiannikov, Silvius Rus, Damian Reeves, Paul Sutter, Sriram Rao, and Jim Kelly. The Quantcast File System. *Proceedings of the VLDB Endowment*, (11):1092–1101, 2013.
28. James S. Plank, Kevin M. Greenan, and Ethan L. Miller. Screaming Fast Galois Field Arithmetic Using Intel SIMD Instructions. In *11th USENIX Conference on File and Storage Technologies (FAST 13)*, pages 298–306, San Jose, CA, 2013. USENIX Association.
29. James S Plank, Jianqiang Luo, Catherine D Schuman, Lihao Xu, Zooko Wilcox-O'Hearn, et al. A Performance Evaluation and Examination of Open-Source Erasure Coding Libraries for Storage. In *Proccedings of the 7th Conference on File and Storage Technologies*, FAST '09, pages 253–265, Berkeley, CA, USA, 2009. USENIX Association.
30. James S Plank, Scott Simmerman, and Catherine D Schuman. Jerasure: A Library in C/C++ Facilitating Erasure Coding for Storage Applications. 2008.
31. KV Rashmi, Mosharaf Chowdhury, Jack Kosaian, Ion Stoica, and Kannan Ramchandran. EC-Cache: Load-Balanced, Low-Latency Cluster Caching with Online Erasure Coding. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, 2016.
32. KV Rashmi, Preetum Nakkiran, Jingyan Wang, Nihar B Shah, and Kannan Ramchandran. Having Your Cake and Eating It Too: Jointly Optimal Erasure Codes for I/O, Storage, and Network-bandwidth. In *FAST*, pages 81–94, 2015.
33. KV Rashmi, Nihar B Shah, Dikang Gu, Hairong Kuang, Dhruba Borthakur, and Kannan Ramchandran. A Solution to the Network Challenges of Data Recovery in Erasure-coded Distributed Storage Systems: A Study on the Facebook Warehouse Cluster. In *HotStorage*, 2013.
34. Irving S Reed and Gustave Solomon. Polynomial Codes Over Certain Finite Fields. *Journal of the Society for Industrial and Applied Mathematics*, 8(2):300–304, 1960.
35. Rodrigo Rodrigues and Barbara Liskov. High availability in dhts: Erasure coding vs. replication. In *International Workshop on Peer-to-Peer Systems*, pages 226–239. Springer, 2005.
36. Maheswaran Sathiamoorthy, Megasthenis Asteris, Dimitris Papailiopoulos, Alexandros G. Dimakis, Ramkumar Vadali, Scott Chen, and Dhruba Borthakur. XORing Elephants: Novel Erasure Codes for Big Data. *Proceedings of the VLDB Endowment*, 6(5):325–336, March 2013.
37. Dipti Shankar, Xiaoyi Lu, and Dhabaleswar K Panda. High-Performance and Resilient Key-Value Store with Online Erasure Coding for Big Data Workloads. In *Distributed Comput-*

ing Systems (ICDCS), 2017 IEEE 37th International Conference on, pages 527–537. IEEE, 2017.

38. Haiyang Shi, Xiaoyi Lu, Dipti Shankar, and Dhabaleswar K Panda. High-Performance Multi-Rail Erasure Coding Library over Modern Data Center Architectures: Early Experiences. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 530–531. ACM, 2018.
39. Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop Distributed File System. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1–10. IEEE, 2010.
40. Dan Terpstra, Heike Jagode, Haihang You, and Jack Dongarra. Collecting Performance Data with PAPI-C. In Matthias S. Müller, Michael M. Resch, Alexander Schulz, and Wolfgang E. Nagel, editors, *Tools for High Performance Computing 2009*, pages 157–173, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.
41. Using Intel® Streaming SIMD Extensions and Intel® Integrated Performance Primitives to Accelerate Algorithms. `https://software.intel.com/en-us/articles/`, 2016.
42. Hakim Weatherspoon and John D Kubiatowicz. Erasure coding vs. replication: A quantitative comparison. In *International Workshop on Peer-to-Peer Systems*, pages 328–337. Springer, 2002.
43. Sage A Weil, Scott A Brandt, Ethan L Miller, Darrell DE Long, and Carlos Maltzahn. Ceph: A Scalable, High-Performance Distributed File System. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 307–320. USENIX Association, 2006.
44. Lihao Xu and Jehoshua Bruck. X-code: MDS Array Codes with Optimal Encoding. *IEEE Transactions on Information Theory*, 45(1):272–276, 1999.
45. Heng Zhang, Mingkai Dong, and Haibo Chen. Efficient and Available In-memory KV-Store with Hybrid Erasure Coding and Replication. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 167–180, Santa Clara, CA, February 2016. USENIX Association.