A Performance Evaluation of Erasure Coding Libraries for Cloud-Based Data Stores
(Practical Experience Report)

Dorian Burihabwa, Pascal Felber, Hugues Mercier, and Valerio Schiavoni(✉)

Université de Neuchâtel, Neuchâtel, Switzerland
{dorian.burihabwa,pascal.felber,hugues.mercier, valerio.schiavoni}@unine.ch

Abstract. Erasure codes have been widely used over the last decade to implement reliable data stores. They offer interesting trade-offs between efficiency, reliability, and storage overhead. Indeed, a distributed data store holding encoded data blocks can tolerate the failure of multiple nodes while requiring only a fraction of the space necessary for plain replication, albeit at an increased encoding and decoding cost. There exists nowadays a number of libraries implementing several variations of erasure codes, which notably differ in terms of complexity and implementation-specific optimizations.

Seven years ago, Plank *et al.* [14] have conducted a comprehensive performance evaluation of open-source erasure coding libraries available at the time to compare their raw performance and measure the impact of different parameter configurations. In the present experimental study, we take a fresh perspective at the state of the art of erasure coding libraries. Not only do we cover a wider set of libraries running on modern hardware, but we also consider their efficiency when used in realistic settings for cloud-based storage, namely when deployed across several nodes in a data centre. Our measurements therefore account for the end-to-end costs of data accesses over several distributed nodes, including the encoding and decoding costs, and shed light on the performance one can expect from the various libraries when deployed in a real system. Our results reveal important differences in the efficiency of the different libraries, notably due to the type of coding algorithm and the use of hardware-specific optimizations.

1 Introduction

Cloud-based storage has seen an impressive growth over the last few years, notably thanks to the availability of affordable solutions from large companies like Dropbox, Google, Amazon, Microsoft, or Apple. While these services mainly target the general public, several companies have also specialized in the development of dedicated solutions offering specific properties in terms of security or dependability, support for deployment on premises, or customized application

© IFIP International Federation for Information Processing 2016
Published by Springer International Publishing Switzerland 2016. All Rights Reserved
M. Jelasity and E. Kalyvianaki (Eds.): DAIS 2016, LNCS 9687, pp. 160–173, 2016.
DOI: 10.1007/978-3-319-39577-7_13

support. There is also a great interest from the scientific community to develop their own solution to finely control and tune a complete Cloud-based stack, be it in terms of communication, processing, or storage.

In the era of "Big Data", the amount of information to manage can quickly become a bottleneck. Storage space is plenty but not infinite, and data must be stored redundantly for reliability purposes. It must therefore be replicated but at the same time remain relatively compact in size. There is therefore a tension between space efficiency and performance, as any form of compression comes with associated processing overhead.

A common approach to address this challenge is to use erasure coding. This technique, already used since the eighties for redundant array of independent disks, allows infrastructure providers to add some redundancy to stored data without the space overhead of full replication, and with relatively low computation costs and sufficient flexibility in how failed disks or missing data can be recovered.

The theory of erasure codes has been developed over decades, and techniques are mature. Yet, practical libraries for performing the associated computations efficiently in software have truly emerged over the last few years, largely driven by the needs from backup services, data centers, and Cloud-based infrastructures in general ("Anything" as a Service).

Seven years ago, Plank *et al.* [14] have conducted a comprehensive performance evaluation of open-source erasure coding libraries available at the time. The focus of the study was on the raw performance of the libraries and the impact of different parameter configurations. Since then, the landscape has significantly evolved, with improved encoders and new libraries, as well as a shift toward integration within software stacks for data centers.

In this experimental study, we want to take a fresh perspective at the state of practice in erasure coding for data storage. Rather than focusing just on the libraries, we are interesting in evaluating them in realistic settings, i.e., within a complete software stack involving multiple clients and server nodes deployed across a data center. This study does also account for the costs related to data serialization and transmission, the overhead associated with the different APIs, the reads and writes to the back-end databases, etc. Therefore, it allows us to quantify the costs of using the libraries to build a complete storage solution, and to observe how the various configuration parameters affect end-to-end performance.

Our findings notably reveal several interesting lessons. First, modern encoding libraries efficiently exploit specific hardware instructions for better performances, and in a cloud-environment it is important to obtain direct access to the CPU's full instruction set. Second, the usage of high-level languages such as Python, which allow for portable and dependable client-side front-ends, to interact with efficient C-based libraries does not cause noticeable overhead. Third, the deployment of encoded data blocks over storage back-ends require practitioners to accommodate much less data than a solution based on replication. Fourth, the reconstruction of missing blocks is a sensibly more costly operation

than the pure decoding when all blocks are available: this is an important factor to take into account in case of disaster-recovery actions to estimate the time to recover missing data.

The remainder of this paper is organized as follows. We first give an overview of erasure coding and related with in Sect. 2. We then describe the different libraries and the storage architecture used as part of this study in Sects. 3 and 4, respectively. We present and discuss experimental results in Sect. 5, and we finally conclude in Sect. 6.

2 Background and Related Work

The objective of error-correcting codes for data storage is to carefully add redundancy to data in order to protect it against corruption when stored on media like DVDs, magnetic tapes or solid-state drives. In these systems, the errors are usually modeled as erasures, meaning that their locations are known. Consider the example shown in Fig. 1, where a coding disk is used to store the XOR ("exclusive or") of k data blocks. If the system realizes that one of the disks has failed, as shown in Fig. 2, it can XOR the healthy disks and recover the failure. This is the maximum decoding capability of this code, and there will be data loss if more than one disk fails.

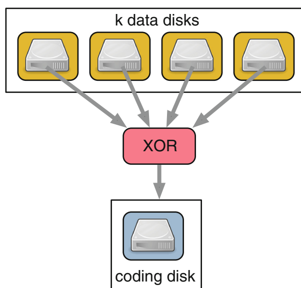

Fig. 1. Redundancy using XOR.

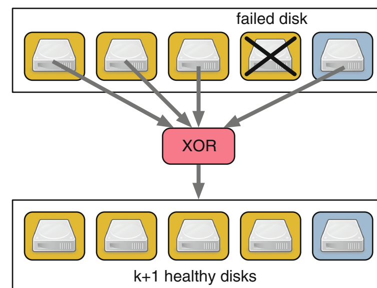

Fig. 2. Failed disk recovered using XOR.

In general, k data blocks are coded to generate $n - k$ coding blocks, as illustrated in Fig. 3. After disk failures, the system will try to decode the original codewords from the healthy disks, like in Fig. 4. The number of recoverable disk failures depends on the code itself.

The most famous class of erasure codes are Reed-Solomon codes (RS), first introduced in 1960 [17]. An (n, k) Reed-Solomon code is a linear block code with dimension k and length n defined over the finite field of n elements. Reed-Solomon codes have many interesting properties. First, they achieve the singleton bound with equality, and thus are maximum distance separable (MDS) [8]. In other words, they can correct up to $n - k$ symbol erasures, i.e., any k of the

n code symbols are necessary and sufficient for decoding. Second, using a large field, they can correct bursts of errors, thus their widespread adoption in storage media where such bursts are common.

Encoding and decoding Reed-Solomon codes is challenging, and optimizing both operations has kept many coding theorists and engineers busy for more than 50 years. There is a large amount of literature on these topics, covering theory (e.g., [5]) and implementation (e.g., [18]), but in a nutshell the best encoding and decoding implementations are quadratic in the size of the data.

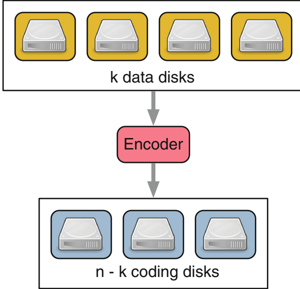

Fig. 3. Generic erasure code encoder.

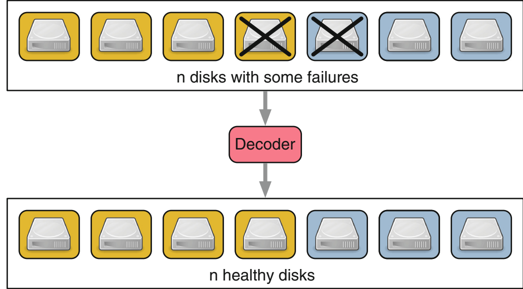

Fig. 4. Generic erasure code decoder.

Reed-Solomon, while storage-efficient, were not originally designed for distributed storage and are somewhat ill-suited for this purpose. Besides their complexity, their main drawback is that they require at least k geographically distributed healthy disks to recover a single failure, followed by decoding of all the codewords with a block on the failed disk. This incurs significant bandwidth and latency costs. This handicap has led to the development of codes that can recreate destroyed redundancy without decoding the original codewords. The tradeoffs between storage overhead and failure repairability is an active area of research [3,11], and there are many interesting theoretical and practical questions to solve. Among other work of interest, NCCloud [2] reduces the cost of repair in multi-cloud storage if one cloud storage provider fails permanently. We also mention the coding work done for Microsoft Azure [1,6], and XOR-based erasure codes [7] in the context of efficient cloud-based file-systems exploiting *rotated* Reed-Solomon codes. RAID-like erasure-coding techniques have been studied in the context of cloud-based storage solutions [7]. Plank *et al.* [14] studied chosen erasure-coding libraries, which has in great part motivated and inspired the present study.

3 Coding Libraries

We tested four different coding libraries in our experimental evaluation. These libraries are considered state-of-the-art and are widely adopted in storage and

networking applications. We describe below the main features of each of them. Table 1 summarizes their principal characteristics.

Liberasurecode. Liberasurecode[1] is an erasure code API library in C that supports pluggable erasure code backends. It supports backends such as jerasure and Intel ISA-L but also provides three erasure codes of its own: a Reed Solomon implementation and two flat XOR implementations. Flat XOR erasure codes [4] are small low-density parity-check (LDPC) codes [10]. With flat XOR codes, each parity element is the XOR of a distinct subset of data elements. Such codes are not maximum distance separable (MDS) and, hence, incur in some additional storage overhead over MDS codes. However, they offer the advantage of additional recovery possibilities, i.e., an element can be recovered using many distinct sets of elements. We evaluate two flat XOR codes constructions, `flat_xor_3` and `flat_xor_4`, that respectively have a Hamming distance of $d = 3$ and $d = 4$.

Jerasure. The Jerasure library,[2] first released in 2007, is one of the oldest and most popular erasure coding library. Jerasure is written in C/C++ and implements several variants of Reed-Solomon and MDS erasure codes (Vandermonde, Cauchy [9], Blaum-Roth, RAID-6 Liberation [12],...). As it has been used in many different projects, Jerasure is also a stable and mature library. It notably provides a rich and well documented API, and has been optimized for speed on modern processors (e.g., by leveraging SIMD instructions since version 2.0). More details about the internals of this library can be found at [15].

Intel ISA-L. Intel Intelligent Storage Acceleration Library (ISA-L)[3] is an implementation of erasure codes optimized for speed on Intel processors [13]. It is written primarily in hand-coded assembler and aggressively optimizes the matrix multiplication operation, the most expensive step of encoding. During the decoding operations, Intel ISA uses a cubic cost Gaussian elimination solver. For our evaluation we the latest version (v2.14).

LongHair. The LongHair library[4] is an implementation of fast Cauchy Reed-Solomon erasure codes in C [16]. It was designed to be portable and extremely fast, and it provides an API flexible enough for file transfer where the blocks arrive out of order.

4 Experimental Cloud-Based Data Store

In order to easily and efficiently evaluate the wide spectrum of coding libraries described previously, and to accelerate the comparison between current and future solutions, we designed and implement a lightweight, yet modular experimental testbed. We describe its components, the internal mechanisms, the deployment infrastructure, and other contextual assumptions in the remainder of this section.

[1] https://bitbucket.org/tsg-/liberasurecode.
[2] http://jerasure.org/jerasure/jerasure.
[3] https://software.intel.com/en-us/storage/ISA-L.
[4] https://github.com/catid/longhair.

Table 1. Summary of encoder names and libraries, support for hardware acceleration (HW), and the description of the algorithms (RS stands for Reed-Solomon).

Encoder	Library	HW	Description
`liberasure_rs_vand`	LibErasure	×	Vandermonde RS
`liberasure_flat_xor_3`	LibErasure	×	Flat-XOR ($d = 3$)
`liberasure_flat_xor_4`	LibErasure	×	Flat-XOR ($d = 4$)
`jerasure_rs_vand`	Jerasure	SIMD (SSSE3), CLMUL	Vandermonde RS
`jerasure_rs_cauchy`	Jerasure	SIMD (SSSE3), CLMUL	Cauchy RS
`isa_l_rs_vand`	Intel ISA-L	SIMD (SSE4), AVX(1/2)	Vandermonde RS
`longhair_cauchy_256`	LongHair	SIMD (SSSE3)	Cauchy RS

4.1 Architecture

In its simplest instantiation, a cloud-based data store that leverages erasure coding comprises the following core components, as depicted in Fig. 5: a storage server ("proxy") that mediates interactions between clients and the data store, an encoder, and a set of storage nodes.

The **proxy** component is the main front-end to the system. While there can be an arbitrary number of proxies for a given data store, we only consider one in our evaluation. The proxy exposes a simple *stateless* REST interface to put and get data blocks. The interface mimics the operating principles of well-established services like Amazon S3. More sophisticated operations, such as operating on subsets of the data blocks for a given file, can be easily integrated. The interactions between the proxy and the clients happen via synchronous HTTP messages over pre-established TCP channels. The proxy dispatches/collects data blocks to/from the encoder service.

The **encoder** component performs the actual processing and transformation of data blocks before they are stored, as well as the reverse decoding operation. The encoder is co-localized within the same host as the proxy to maximize

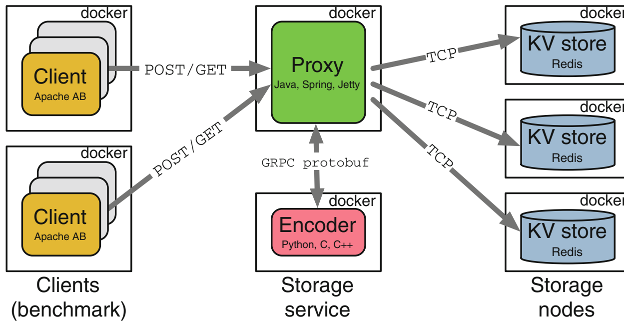

Fig. 5. Architecture of the experimental testbed.

throughput and avoid bottlenecks induced by high pressure on the network stack. To increase the flexibility of our testbed, our encoder provides a plugin mechanism to dynamically load and swap different coding libraries. This mechanism relies on a platform-independent transport mechanism (using `protobuf`) and a stable interface between the proxy and the encoder. The encoder interface currently exposes three main operations: `encode`, `decode`, and `reconstruct`.

Once the blocks have been encoded, they are sent by the proxy to the **storage nodes**. Each storage node is independent from the others and all the interactions are mediated by the proxy. Blocks are dispatched to storage nodes using an explicit placement strategy: the proxy ensures that encoded block parts are spread to distinct nodes so that the failure of one node results in the loss of one part. Upon read, the proxy contacts a random subset of storage nodes of minimal size to reconstruct the requested block.

The **clients** are separate processes running on different nodes in the same data center. They read and write data by contacting the proxy following access patterns defined as part of the workloads.

4.2 Implementation

We used different languages and technologies to implement our testbed and integrate with the open-source erasure coding libraries. Our implementation choices have been largely driven by performance and simplicity considerations, as well as by constraints from the evaluated libraries.

The proxy component is implemented in Java and exploits the exporting facilities of the *Spring Boot* framework[5] (v1.3.1) to leverage industrial-grade application servers. The proxy handles `POST` and `GET` requests via the embedded Jetty web-server.

The encoder component is implemented in Python to facilitate the integration with the PyEClib[6] library (v1.2), the reference Python binding for `liberasure`. This library implements wrappers to uniformly access several encoding libraries.

The open-source libraries under evaluation are implemented in C/C++ (e.g., `liberasure`, `JErasure`, `LongHair`) or a mix of C and hand-written Assembly (e.g., `Intel ISA-L`). These implementation choices lead to the best performances and can take advantage of hardware acceleration, as in the case of `Intel ISA-L` that exploits built-in SIMD CPU instructions. We call the libraries via a common access layer and software wrappers implemented in Python. Python provides an easy mean to bind to such libraries via its built-in support for native code. For completeness, we also evaluate the overhead of using an interpreted language such as Python in our experimental validation. The suite of macro-benchmarks leverages Apache Bench[7] (v2.3) to measure the maximum throughput and per-request latencies. The storage nodes run on top of Redis[8] (v3.0.7), a lightweight

[5] https://projects.spring.io/spring-boot/.
[6] https://pypi.python.org/pypi/PyECLib.
[7] https://httpd.apache.org/docs/2.4/programs/ab.html.
[8] http://redis.io.

yet efficient in-memory key-value store. Redis tools provide easy-to-use probing mechanisms (e.g., the `redis-cli` command-line tool) to retrieve the current memory occupied by the stored key-value pairs. We exploit these tools to calculate the storage overhead of the encoded blocks. Our deployment machinery allows us to scale at will all the mentioned components. To do so, we rely on the tools offered by the *Docker*[9] ecosystem (v1.6) and its *Compose*[10] orchestration tool (v1.5.3).

5 Experimental Results

This section presents our extensive evaluation of the previously described coding libraries. We first describe the settings of our evaluation, then we test in isolation the coding libraries via a set of micro-benchmarks, and we finally evaluate the libraries in realistic settings.

5.1 Evaluation Settings

We deploy and conduct our experiments over a cluster of machines interconnected by a 1 Gb/s switched network. Each physical host features 8-Core Xeon CPUs and 8 GB of RAM. We deploy virtual machines (VM) on top of the hosts.

The KVM hypervisor, which controls the execution of the VM, is configured to expose the physical CPU to the guest VM and Docker container by mean of the `host-passthrough`[11] option. This VM configuration allows certain coding libraries (e.g., Intel ISA-L) to exploit ad-hoc CPU instruction sets. In this evaluation, we do not account for any GPU acceleration. The VMs leverage the `virtio` module for better I/O performances.

We deploy one Docker container on each VM without any memory restriction to minimize interferences due to co-locations and maximize performances.

5.2 Micro-benchmark

Our first set of experiments evaluate the throughput of the coding libraries for increasing block sizes of 4 MB, 16 MB and 64 MB. In this scenario, the libraries are tested in isolation via specialized clients that send a continuous stream of data blocks to encode or decode.

For each library we execute 10,000 times the `encode` function and show the average and standard deviation results. All the Reed-Solomon libraries are configured with $k = 10$ and $m = 4$, a typical configuration used in modern data centers (e.g., at Facebook [19]). To approach a similar configuration, the Flat XOR libraries are set to $k = 10$ and $m = 5$. For reference purposes, we compare against a baseline `striping` encoder/decoder that simply splits the data in the

[9] https://www.docker.com.
[10] https://docs.docker.com/compose/.
[11] http://www.linux-kvm.org/page/Tuning_KVM.

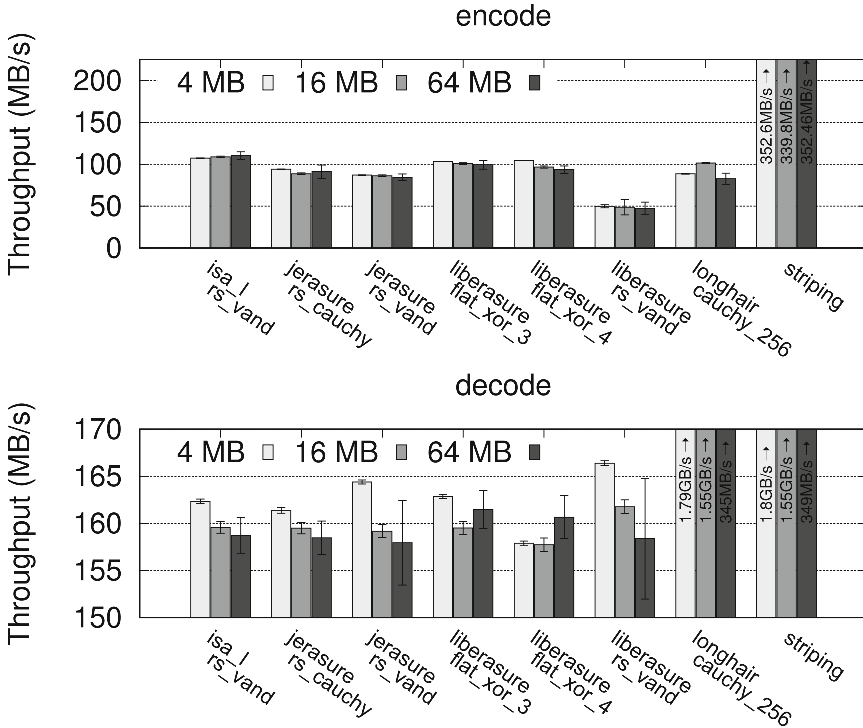

Fig. 6. Micro-benchmark: `encode` (top) and `decode` (bottom) throughput for several coding libraries and block sizes.

requested number of blocks (typically one block per stripe) and immediately returns them to the client without any further processing.

Figure 6 presents our results for encoding (top) and decoding (bottom). We notice that `liberasure_rs_vand` is the slowest in the encoding phase, achieving at most 52.35 MB/s for a 4 Mb block size. The Jerasure implementation of the same coding technique (`jerasure_rs_vand`) and Intel's `isa_l_rs_vand` perform *twice* as fast for the same block size, respectively up to 87 MB/s and 107 MB/s.

We can explain the performance gap between different implementations of the same coding techniques by two main reasons: (1) the longer foray of such libraries in the open-source community (the original design of Jerasure dates back to 2007) thus benefiting from several contributions and code scrutiny, and (2) native support for hardware acceleration for the Intel ISA and Jerasure libraries.

Finally, `longhair_cauchy_256` outperforms the other implementations for any block size. Indeed, not only is its implementation based on the Jerasure source code, but it embeds carefully hand-crafted low-level optimizations (e.g., selection of the minimal Cauchy matrix, faster matrix bootstrap, etc.).

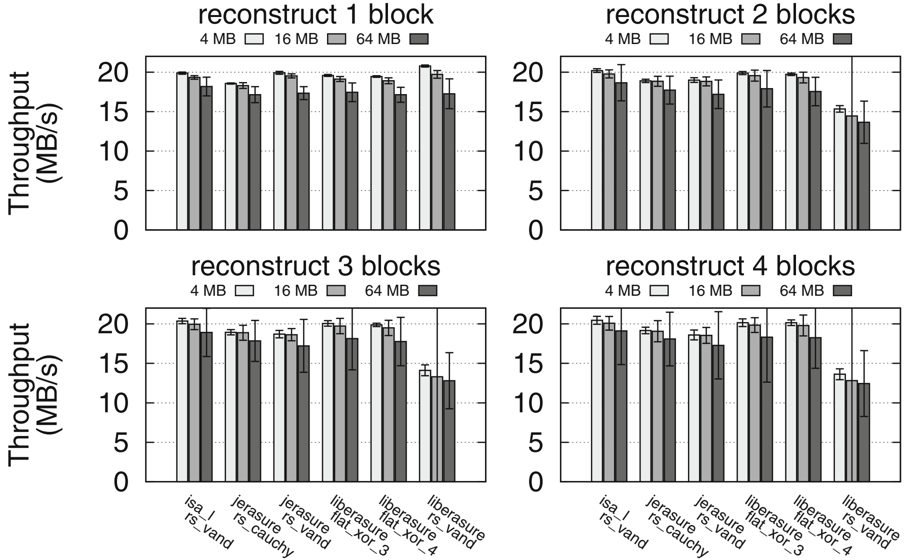

Fig. 7. Micro-benchmarks: throughput of `reconstruct` for increasing number of missing blocks and block sizes.

In the decoding scenario, the `decode` function is fed with all the available blocks. As expected, when all the blocks are available, the libraries can decode very efficiently, achieving throughputs that are never below 157 MB/s for any block size. For example, `liberasure_flat_xor_4` achieves a 158.13 MB/s throughput with 4 MB blocks, and `jerasure_rs_cauchy` reaches 164.87 MB/s. The highly optimized `longhair_cauchy_256` achieves results that are orders-of-magnitude better also in decoding (up to 1.79 GB/s for 4 MB blocks).

Finally, Fig. 7 shows the cost of reconstructing missing blocks. We present the achieved throughput of the coding libraries in reconstructing from 1 to 4 missing blocks (from top-left to bottom-right). Figure 7 presents the average throughput for 100 executions. Notice how `liberasure_rs_vand` achieves the best result (20.77 MB/s) in reconstructing 1 missing block with 4 MB block sizes but steadily decreases with bigger block sizes and more to reconstruct. This result confirms the measures of the same library in pure decoding shown previously in Fig. 6. The other libraries perform consistently across the spectrum of parameters, and all operate between 17.15 MB/s and 19.19 MB/s. These results need to be taken carefully into account to decide which is the best fitting library to adopt in a cloud setting.

We performed a breakdown analysis of the computing times for each of the microbenchmarks. The goal of this analysis is to verify that the cost of using a high-level language such as Python did not hinder our results and thus negatively impacted on the observed performances. We exploit the `cProfile` module[12] to

[12] https://docs.python.org/2/library/profile.html.

profile the execution of the `encode`, `decode`, and `reconstruct` microbenchmarks, and to gather profiling statistics. Indeed, the CPU spends almost the totality of the execution time (always more than 99 %) in the native code of the encoding libraries. These results confirm the choice of Python as having near-zero impact on the overall performances, while providing major benefits in ease of programming, deployment, and availability of open-source libraries.

5.3 Macrobenchmark

In this section we evaluate the coding libraries in a more complex scenario that involves a large-scale storage infrastructure service. First we focus on the observed per-request latency as measured by an external client that stores files into the system. The client is implemented on top of the Apache `ab` benchmark tool. It issues `POST` requests using a randomly generated key and payload, which are sent to the proxy and eventually stored into the Redis storage nodes.

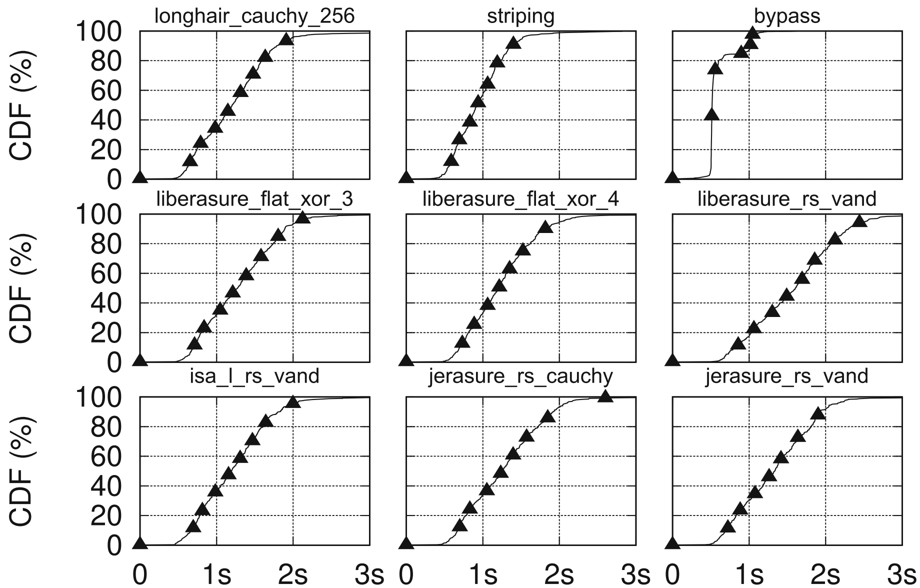

Fig. 8. Latency distribution of 500 requests measured by Apache `ab` client.

Figure 8 shows the cumulative distribution function (CDF) of the latencies as observed by the client. These results include two additional variants that concretely avoid any encoding actions: `striping` and `bypass`. The `striping` variant simply splits the file into the desired number of blocks and immediately returns them to the client. We implemented this solution as an additional backend to the PyECLib library. The `bypass` technique allows to circumvent any communication overhead between the proxy and the encoder components (see

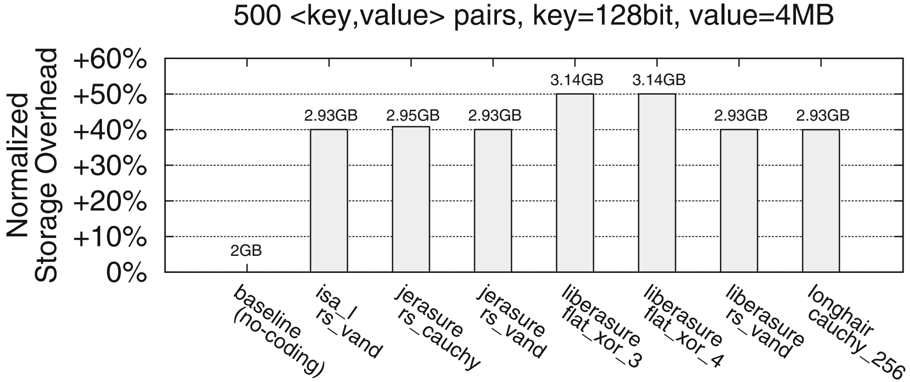

Fig. 9. Storage overhead.

Fig. 5). The difference between the `bypass` and `striping` indicate the price one pays to send the files back and forth between the proxy and the encoder. We observe that `isa_l_rs_vand` and `longhair_cauchy_256` achieve the best overall performances, with the 99th percentile of the latencies below 2.645 s and 2.482 s respectively, and the median latencies as low as 1.192 s and 1.133 s.

We conclude our experimental evaluation by comparing the storage overhead induced by the choice of a given coding library. Figure 9 presents our results. The client sequentially stores 500 files of 4 MB each, for a total of 2 GB of data. The baseline results indicate the cost of storing the files without any form of coding. On the y-axis we show the storage overhead normalized against the baseline cost, while for each library we indicate the total space requirements. In our experiments, the Flat XOR erasure codes are on average 8 % more demanding than the other codes: they require a total of 3.14 GB of storage space (corresponding to a +63 % of the original data).

6 Conclusion

We have studied and compared, in this practical experience report, the performance of several open-source erasure coding libraries that are widely used to implement error correction in distributed systems. These libraries notably differ in terms of coding algorithms, implementation quality, and hardware-specific optimizations. Unlike the seminal study of Plank *et al.* [14] published seven years ago, we focus here on the latest generation of coding libraries when used in realistic settings for cloud-based storage, and deployed on modern hardware inside a data centre.

We conducted a wide range of experiments with these libraries to not only measure their raw speed at encoding and decoding data, but also evaluate their performance when used to store and retrieve actual content. Our observations notably highlight the importance of specific hardware instructions such as SIMD

to improve performance, the negligible overhead of using coding libraries in high-level languages like Python, the good space efficiency of erasure codes for fault-tolerant storage, and the relatively high cost of the reconstruction of missing blocks as compared to regular decoding operations.

The objective of this experimental study was to evaluate and compare existing solutions, rather than develop original coding methods. We hope that it will bring valuable insights and guidance to other researchers interested in using erasure coding for data storage.

References

1. Calder, B., Wang, J., Ogus, A., Nilakantan, N., Skjolsvold, A., McKelvie, S., Xu, Y., Srivastav, S., Wu, J., Simitci, H., Haridas, J., Uddaraju, C., Khatri, H., Edwards, A., Bedekar, V., Mainali, S., Abbasi, R., Agarwal, A., Fahim ul Haq, M., Ikram ul Haq, M., Bhardwaj, D., Dayanand, S., Adusumilli, A., McNett, M., Sankaran, S., Manivannan, K., Rigas, L.: Windows Azure Storage: a highly available cloud storage service with strong consistency. In: Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles, SOSP 2011, pp. 143–157. ACM, New York (2011)
2. Chen, H.C., Hu, Y., Lee, P.P., Tang, Y.: NCCloud: a network-coding-based storage system in a cloud-of-clouds. Trans. Comput. **63**(1), 31–44 (2014)
3. Dimakis, A., Godfrey, P., Wu, Y., Wainwright, M., Ramchandran, K.: Network coding for distributed storage systems. IEEE Trans. Inf. Theory **56**(9), 4539–4551 (2010)
4. Greenan, K.M., Li, X., Wylie, J.J.: Flat XOR-based erasure codes in storage systems: constructions, efficient recovery, and tradeoffs. In: 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST), pp. 1–14. IEEE (2010)
5. Guruswami, V., Sudan, M.: Improved decoding of Reed-Solomon and algebraic-geometry codes. IEEE Trans. Inf. Theory **45**(6), 1757–1767 (1999)
6. 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 the USENIX Annual Technical Conference, USENIX ATC, pp. 15–26 (2012)
7. Khan, O., Burns, R.C., Plank, J.S., Pierce, W., Huang, C.: Rethinking erasure codes for cloud file systems: minimizing I/O for recovery and degraded reads. In: Proceedings of the 7th Conference on File and Storage Technologies, FAST 2012, p. 20. USENIX Association (2012)
8. Lin, S., Costello, D.J.: Error Control Coding, 2nd edn. Pearson Prentice Hall, Englewood Cliffs (2004)
9. Luby, M., Zuckermank, D.: An XOR-based erasure-resilient coding scheme. Technical report (1995)
10. Luby, M.G., Mitzenmacher, M., Shokrollahi, M.A., Spielman, D.A., Stemann, V.: Practical loss-resilient codes. In: Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing, pp. 150–159. ACM (1997)
11. Oggier, F.E., Datta, A.: Self-repairing codes - local repairability for cheap and fast maintenance of erasure coded data. Computing **97**(2), 171–201 (2015)
12. Plank, J.S.: The raid-6 Liber8Tion code. Int. J. High Perform. Comput. Appl. **23**, 242–251 (2009)
13. Plank, J.S., Greenan, K.M., Miller, E.L.: Screaming fast Galois field arithmetic using Intel SIMD instructions. In: FAST, pp. 299–306 (2013)

14. Plank, J.S., Luo, J., Schuman, C.D., Xu, L., Wilcox-O'Hearn, Z.: A performance evaluation and examination of open-source erasure coding libraries for storage. In: Proceedings of the 7th Conference on File and Storage Technologies, FAST 2009, pp. 253–265. USENIX Association, Berkeley (2009)
15. Plank, J.S., Simmerman, S., Schuman, C.D.: Jerasure: A library in C/C++ facilitating erasure coding for storage applications-version 1.2. Technical report, Technical Report CS-08-627, University of Tennessee (2008)
16. Plank, J.S., Xu, L.: Optimizing Cauchy Reed-Solomon codes for fault-tolerant network storage applications. In: 2006 Fifth IEEE International Symposium on Network Computing and Applications, NAC 2006, pp. 173–180. IEEE (2006)
17. Reed, I.S., Solomon, G.: Polynomial codes over certain finite fields. J. Soc. Ind. Appl. Math. **8**(2), 300–304 (1960)
18. Sankaran, J.: Reed Solomon decoder: TMS320C64x implementation. Application Report SPRA686, Texas Instruments (2000)
19. Sathiamoorthy, M., Asteris, M., Papailiopoulos, D., Dimakis, A.G., Vadali, R., Chen, S., Borthakur, D.: XORing elephants: novel erasure codes for big data. Proc. VLDB Endow. **6**(5), 325–336 (2013)