ARC: An Automated Approach to Resiliency for Lossy Compressed Data via Error Correcting Codes

Dakota Fulp
dakotaf@clemson.edu
Holcombe Department of Electrical and Computing Engineering
Clemson, South Carolina, USA

Alexandra Poulos
alpoulo@clemson.edu
Holcombe Department of Electrical and Computing Engineering
Clemson, South Carolina, USA

Robert Underwood
robertu@clemson.edu
School of Computing
Clemson, South Carolina, USA

Jon C. Calhoun
jonccal@clemson.edu
Holcombe Department of Electrical and Computing Engineering
Clemson, South Carolina, USA

ABSTRACT

Progress in high-performance computing (HPC) systems has led to complex applications that stress the I/O subsystem by creating vast amounts of data. Lossy compression reduces data size considerably, but a single error renders lossy compressed data unusable. This sensitivity stems from the high information content per bit in compressed data and is a critical issue as soft errors that cause bit-flips have become increasingly commonplace in HPC systems. While many works have improved lossy compressor performance, few have sought to address this critical weakness.

This paper presents ARC: Automated Resiliency for Compression. Given user-defined constraints on storage, throughput, and resiliency, ARC automatically determines the optimal error-correcting code (ECC) configuration before encoding data. We conduct an extensive fault injection study to fully understand the effects of soft errors on lossy compressed data and how to best protect it. We evaluate ARC's scalability, performance, resiliency, and ease of use. We find on a 40 core node that encoding and decoding demonstrate throughput up to 3730 MB/s and 3602 MB/s. ARC also detects and corrects multi-bit errors with a tunable overhead in terms of storage and throughput. Finally, we display the ease of using ARC and how to consider a systems failure rate when determining the constraints.

CCS CONCEPTS

• **Theory of computation → Data compression**; • **Software and its engineering → Software fault tolerance**.

KEYWORDS

error-bounded lossy compression, SZ, ZFP, soft error, error correcting codes, error propagation, silent data corruption

HPDC '21, June 21–25, 2021, Virtual Event, Sweden

ACM ISBN 978-1-4503-8217-5/21/06…$15.00
https://doi.org/10.1145/3431379.3460638

ACM Reference Format:
Dakota Fulp, Alexandra Poulos, Robert Underwood, and Jon C. Calhoun. 2021. ARC: An Automated Approach to Resiliency for Lossy Compressed Data via Error Correcting Codes. In *Proceedings of the 30th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '21), June 21–25, 2021, Virtual Event, Sweden.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3431379.3460638

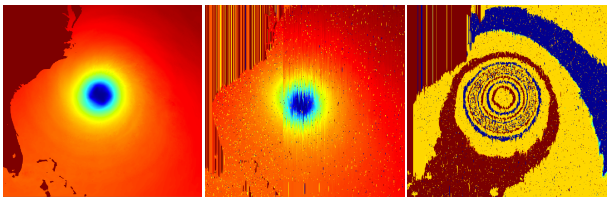
1 INTRODUCTION

High-performance computing (HPC) systems have become a critical part of scientific discoveries and have made solving previously intractable problems possible. However, HPC applications are rapidly producing vast amounts of data, causing significant bottlenecks in the I/O subsystem [14, 24]. Scientists often use various types of compression to reduce the size of these datasets.

Lossless compressors, such as GZip [11] and ZStd [5], reduce data size with no loss in precision through statistical modeling and data value mapping. However, due to the high entropy of the mantissa bits in HPC floating-point data, they suffer from suboptimal compression ratios [30].

Lossy compressors, such as the industry-standard compressors SZ [7] and ZFP [20], achieve higher compression ratios by reducing data precision, making them ideal for HPC floating-point data [3, 4, 7, 10, 18, 20, 36]. The user controls the precision reduction through an error-bounding value and mode, allowing them to set the tolerable amount of error to introduce.

The likelihood of encountering soft errors exist for all HPC applications and data. Using Sridharan et al.'s work, in Section 6.4, we calculate soft error failures occur every 1.9 days on the Cielo HPC system [31, 33]. However, this rate does not include undetected soft errors, which cause silent data corruption (SDC). Figure 1 demonstrates the impact of a single-bit soft error at two different bit locations in the SZ lossy compressed Hurricane Isabel dataset when using an error bound of 0.1. In Figure 1(b), the error occurs in bit 400,005 of the compressed data, while in Figure 1(c), the error occurs in bit 465,840. When decompressed, both resulting datasets have a high percent of incorrect elements, defined as the number of data points whose error violates the user set error bound. Specifically, Figure 1(b) and Figure 1(c) have 49.6% and 99.4% incorrect elements. While lossy compression demonstrates a severe sensitivity to soft



(a) Control (b) 49.6% Incorrect (c) 99.4% Incorrect

Figure 1: Effect of a single-bit soft error at two different locations in the Hurricane Isabel pressure dataset compressed with SZ-ABS and an error bound of $\epsilon = 0.1$.

errors, few works have aimed to understand the effects on and protect lossy compressed data from these errors [15, 16, 22, 28].

In this paper, we obtain a better understanding of bit corruption's impact on lossy compressed data through an extensive fault injection study. Using these findings, we develop ARC: Automated Resiliency for Compression, which automatically secures lossy compressed data fidelity using error-correcting codes (ECC) while following user constraints on storage, throughput, and resiliency. Without ARC, a single soft error renders lossy compressed data unusable; by using ARC, error propagation and SDC are no longer likely. Specifically, our novel contributions are as follows:

- We conduct an extensive fault injection study on two of the most notable lossy compressors, SZ [7] and ZFP [20]. Our findings show the effects of a single soft error range in severity based on the location of the bit corrupted and the lossy compression algorithm used.
- Using the results of our fault injection study as a guide, we develop ARC to preserve data fidelity in the presence of soft errors. User-defined constraints on storage, throughput, and resiliency allow ARC to choose and apply the optimal ECC configuration to secure lossy compressed data while still being flexible for use in various HPC domains.
- We evaluate ARC's abilities on the criteria of scalability, performance, resiliency, and ease of use. We find ARC meets user constraints on storage, throughput, and resiliency while demonstrating encoding and decoding throughput of 3730 MB/s and 3602 MB/s, respectively, on a 40 core node. We also find that ARC requires minimal effort to integrate and protects from multi-bit errors while having a tunable storage and throughput overhead.

2 BACKGROUND

2.1 Lossy Compression Algorithms

HPC scientists often use compression to reduce data sizes and resolve I/O bottleneck issues [3, 4, 7, 10, 18, 36]. Lossless compressors, such as GZip [11] and ZStd [5], compress data with no accuracy loss. However, they are suboptimal for HPC floating-point data as they can only achieve compression ratios of 1× to 4× [30]. Lossy compressors, such as the industry-standard SZ [7] and ZFP [20], use data approximation and partial data omission to represent the original data at a lower precision. High compression ratios are possible by introducing user-bounded error, and the effect of this error has been studied widely across various domains [3, 21, 23, 26].

2.1.1 SZ. uses a block-wise prediction-based compression model with three main steps [17]. These steps include data point predictions, linear-scale quantization to convert the data to integer codes, and compression of the integer codes using lossless compression.

In this paper, we examine three error bounding modes of SZ. Using the absolute (SZ-ABS) mode, each data value uses the same user-specified error bound, ensuring uniform precision for all values. Using the point-wise relative (SZ-PWREL) mode, the product of a data value and the set error bound determine its specific error bound. This mode assumes larger data values can tolerate more error while smaller ones require extra precision. Using the peak-signal-to-noise ratio (SZ-PSNR) mode, the data is compressed to retain a minimum PSNR rating. PSNR is a prevalent metric that assesses data distortion when using lossy compression. This mode focuses on overall data integrity over any single value.

2.1.2 ZFP. uses a block-wise transformation-based compression model with three main steps [20]. These steps include fixed-point representation conversion, near orthogonal block transformations, and embedded coding to encode each bit-plane.

In this paper, we examine two error bounding modes within ZFP. Using ZFP's accuracy (ZFP-ACC) mode, each data value uses the same user-specified error bound, similar to SZ-ABS. Conversely, ZFP's fixed-rate (ZFP-Rate) mode divides the data into 4^d sized blocks, where d is the data dimensionality. The product of the user-defined rate and 4^d determines each block's compressed size with a lower rate leading to higher compression and less precision in the data. Out of all modes we examine, ZFP-Rate mode is the only one that supports random access as it decouples dependencies between 4^d sized blocks of data. However, as a trade-off, ZFP-Rate mode cannot bound the introduced error or achieve the other algorithms' higher compression ratios.

2.2 Error Correcting Codes

Error-correcting codes (ECC) allow for detection and, depending on the ECC used, correction of one or more errors by adding redundancies to data. ECC is a popular approach to protecting data because it is application and data agnostic, and it requires significantly less overhead compared to keeping multiple copies of a dataset. For this reason, we choose to use ECC to protect lossy compressed data from errors. In this work, we use four different ECC algorithms.

Parity codes use a single bit to ensure an even number of bits are set to 1 in the data. After applying parity, if an odd number of bits set to 1 occurs, the parity bit becomes incorrect and alerts to the fault. Parity codes have little overhead and detect all odd multi-bit errors but cannot correct errors or detect even multi-bit errors.

Hamming codes are able to detect and correct a single bit in error using a parity check matrix and a syndrome to locate the corrupt bit. The number of bits used to protect each data block depends on the block's size, with the overhead decreasing as block size increases.

SEC-DED (Single-Error Correct Double-Error Detect) codes are a variation of Hamming codes that include an additional parity bit. Similar to standard Hamming codes, SEC-DED codes are able to

detect and correct single-bit errors. The additional parity bit allows them to also detect double-bit errors.

Reed-Solomon codes break data into blocks, called data devices, and use them to create parity code devices. A Reed-Solomon code can correct m corrupted devices, where m equals the number of earlier produced code devices. Reed-Solomon encoding introduces high amounts of overhead but is ideal for handling burst errors.

3 RELATED WORK

3.1 Fault Injection Studies

Soft errors are becoming increasingly commonplace in HPC systems, and many previous studies have investigated their impacts [8, 13, 27, 29, 31–34]. While factors such as node temperature and location affect the possibility of soft errors [31], failure rates and root causes vary among systems. With this in mind, few studies have explored the effects of such errors on compressed data specifically.

In 2017, Avramenko et al. analyzed the effects of soft errors to gauge their impact on lossless compressed data [1]. In 2020, Li et al. analyzed SZ's internal subroutines to discover their susceptibility to SDC [15]. In 2020, Tao et al. used error distribution-based fault models to simulate lossy compression errors in HPC applications to discover the effects of the errors on the application's results [28]. While each of these works study errors in compressed data, they focus on lossless compression, internal subroutines, or application-specific effects of lossy compression errors. This study is broader as we do not focus on protecting a single lossy compressor, and we also perform an extensive soft error injection study. To the best of our knowledge, our work is the first to offer this perspective. When combined with prior works, this work enables a more complete analysis of compression soft error sensitivities.

3.2 Error Resiliency Works

As HPC soft errors cannot be prevented entirely, resiliency is vital, and many previous studies focus on this concept [2, 6, 9, 12, 37]. While their methods vary, the goal of producing error-resilient systems is central throughout these efforts. However, only a few studies have aimed to create more error-resilient compressed data.

In 2005, Nguyen et al. developed a fault-tolerant error-detecting system for the JPEG 2000 image compression standard [22]. In 2020, Li et al. developed an SDC resilient version of SZ to protect SZ's internal operations from errors [15]. In the same year, they also developed a series of data-analytic-based fault tolerance methods for applications using lossy compression [16]. Similarly, ARC aims to protect lossy compressed data but in a decoupled black-box way that does not require extra knowledge of or changes to existing lossy compressors and is not bound to any current or future algorithm.

4 SOFT ERROR EFFECTS ON LOSSY COMPRESSED DATA

Soft errors are a potential threat to all HPC applications and data, but this is especially true for lossy compressed data as a single soft error renders the data unusable, as we show in Figure 1. Moreover, HPC data spends long durations compressed, which further compounds its sensitivity to memory soft errors. To develop a framework to protect this data effectively, we must fully understand how soft errors interact with lossy compressed data. In particular, we aim to examine how soft errors in lossy compressed data lead to error propagation and reductions in data quality.

4.1 Experimental Design

4.1.1 ***Compressors.*** In this paper, we evaluate the resiliency of two leading lossy compressors: SZ 2.1.8.1 [17] and ZFP 0.5.5 [20]. As we state in Section 2.1, for SZ, we examine three error bounding modes, and for ZFP, we examine two. We use an error bound of $\epsilon = 0.1$ for SZ-ABS, SZ-PWREL, and ZFP-ACC, a PSNR rating of 90 for SZ-PSNR, and a rate of 8 for ZFP-Rate. We chose these bounds to maintain consistency with other works [17]. We also adjust the error bounds to test compression ratios of 50×, 25×, 13×, and 7×.

To simplify all interactions with SZ and ZFP, we leverage the compression abstraction library LibPressio [35]. LibPressio abstracts user interactions with various lossless and lossy compressors while normalizing their outputs.

4.1.2 ***Datasets and System.*** In our trials, we use three industry HPC datasets from SDRBench[1] that are commonly used in compression studies [7, 17]. First, the CESM dataset is a global climate model sponsored by the National Science Foundation (NSF) and U.S Department of Energy. From this dataset, we use the 25.82 MB 2D CLDLOW data. Next, the Hurricane Isabel dataset reflects the 2003 hurricane visualization produced by the Weather Research and Forecast model, courtesy of NCAR and the NSF. From this dataset, we use the 100 MB 3D pressure data. Finally, the NYX simulation studies dark matter in the universe, and from this dataset, we use the 536 MB 3D temperature data. We use these datasets as they differ in size and come from a range of scientific domains.

We run all fault injection trials on nodes containing an Intel Xeon E5-2665 16 core CPU with 128GB of memory.

4.1.3 ***Evaluation Metrics.*** In each experiment trial, we flip a single bit of the compressed dataset stored in application memory. As exhaustive testing for a single dataset results in between 1 million and 2.7 trillion trials, we use a uniform sampling approach to select target bits, thus making the study tractable. We conduct a 1%, 0.1%, and 0.01% uniform sampling of CESM, Hurricane Isabel, and NYX, respectively, with these sampling sizes chosen based on data size.

After flipping a bit, we attempt to decompress the data and record the decompression return status. These statuses allow us to determine the percentage of trials that continue to use the defective data and, in turn, lead to error propagation and silent data corruption (SDC). If the data decompresses, we analyze the resulting data to quantify its integrity and accuracy with regard to the original data.

We employ four metrics to analyze the data integrity. First, we record the *percent of incorrect elements*, defined as the number of data values whose error violates the set error bound. This enables us to assess the extent of error propagation in the data. We collect this metric in all trials except those using the SZ-PSNR mode, as PSNR does not bound error at each value. Second, we record the *maximum absolute difference* between the corrupted and original dataset. Without corruption, this value is within the set error bound. Using this metric, we quantify the extent to which the data corruption violates the error bound. Third, we record the *PSNR* of the

[1] https://sdrbench.github.io/

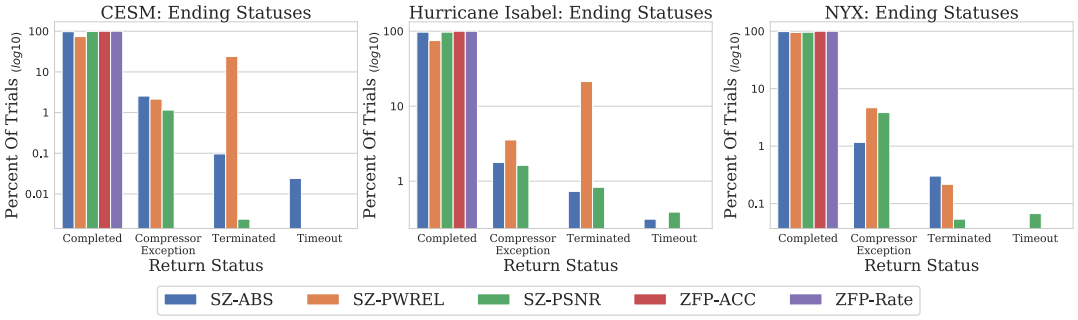


Figure 2: Distribution of return statuses for all fault injection trials.

corrupted data to compare its data quality to the original. When calculating PSNR, we first compute the root-mean-squared error (RMSE), as seen in Equation 1, where N is the data size, d_i is each data point of the original data, and d_i' is the corresponding data point in the corrupted data. Using this, we compute the PSNR, as seen in Equation 2, where d_{max} and d_{min} are the maximum and minimum values of the original data. Finally, we record the corrupted data *decompression bandwidth* to expose any changes in throughput caused by the error. Using these metrics, we discover all changes in decompression and the resulting data quality.

$$RMSE = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(d_i - d_i')^2} \tag{1}$$

$$PSNR = 20 \times log_{10}(\frac{d_{max} - d_{min}}{RMSE}) \tag{2}$$

4.2 Error Effects on Decompression Process

Using our results, we start by examining all return statuses and group them into four categories: *Completed*, *Compressor Exception*, *Terminated*, and *Timeout*. In *Completed* trials, the data successfully decompresses with the error present, leading to a high risk of error propagation and SDC. *Compressor Exception* trials end when the compressor throws an exception due to the error invalidating the data. *Terminated* trials end less gracefully and instead crash the application suddenly due to the corrupted data. Both *Compressor Exception* and *Terminated* trials do not lead to error propagation or SDC, but instead lead to lost productivity. Lastly, *Timeout* trials mean the trial spent 3× the average time trying to decompress and are due to corruptions in decompression loop controlling metadata.

Figure 2 shows the distribution of return statuses of all trials. On average, we find 95.28% of all trials *Completed* while the other 4.72% of trials fell into the other three categories. It is worth noting that Avramenko et al. found similar results when studying lossless compressed program variables, with 41.9% – 96.1% of fault injections leading to silent data corruption [1]. Comparing our results from the different datasets, we see similar results with some outliers, which we attribute to the sample space we test on. These results are troubling as *Completed* trials do not acknowledge soft errors and have a high probability of error propagation and SDC. More troubling is 100% of trials using ZFP *Completed* indicating ZFP will never inherently catch soft error data corruption. Therefore, if soft errors impact lossy or lossless compressed data, SDC may potentially alter and degrade the data quality.

4.3 Error Effects on Error Bounding Capability

We next gauge the impact a corruption has on a compressor's error bounding ability by focusing only on the *Completed* trials. To determine whether the decompressed values violate the error bound, we calculate the difference between each corresponding pair of values in the original and decompressed dataset with corruption. We quantify the frequency for all trials but only show CESM trials in Figure 3 as the other datasets display similar results.

Figure 3(a), (b), and (c) show the results for SZ-ABS, SZ-PWREL, and ZFP-ACC, respectively. From this data, we find a single error in the compressed data leads to an average of ~10% incorrect elements. When using SZ-ABS, this percentage ranges from 0.01 – 80%, when using SZ-PWREL ranges from 0.03 – 64.4%, and when using ZFP-ACC ranges from 0.002 – 53.6%. This is problematic as, on average, a single soft error propagates to 10% of data values leading to SDC. If used, the SDC propagates and corrupts future calculations.

Our findings confirm that certain bits are more important in the decompression process and are used to reconstruct numerous data values. If corrupted, these bits lead to corruption in the bulk of the data. If an error causes order-of-magnitude shifts in the data this

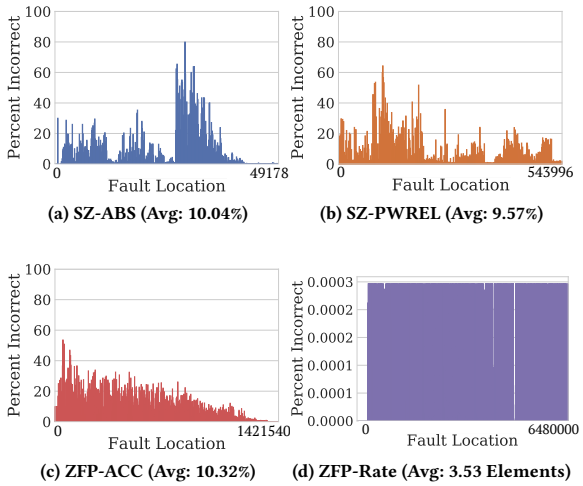


Figure 3: CESM: Percent of elements which violate the set error bound per fault injection location.

could alert the user to the corruption, but this cannot be relied on as order-of-magnitude shifts do no happen in all cases.

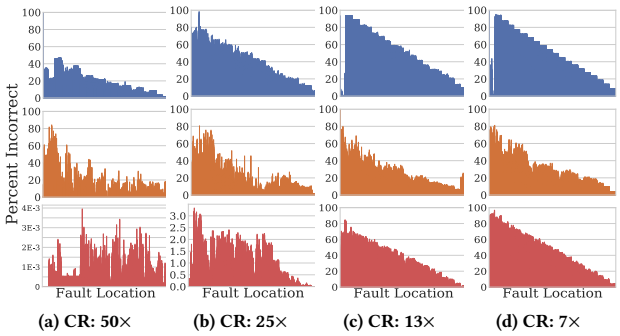
Unlike the other three modes, we find that when using ZFP-Rate a single error leads to an average of only 3.53 data values violating the error bound when decompressed and ranges from 0 – 16 data values. This can be seen in Figure 3(d) and demonstrates the resiliency block compression algorithms have due to the removed dependencies between data blocks. ZFP-Rate mode removes dependencies by dividing the data into 4^d sized blocks, which prevents the error from propagating outside of a given block. However, as a result, it only achieves a compression ratio of 4× while SZ-ABS, SZ-PWREL, and ZFP-ACC achieve compression ratios of 500×, 45×, and 17×.

4.4 Error Effects at Various Levels of Loss

To fully understand error effects on data compressed to different levels, we run additional experiments with each dataset and the SZ-ABS, SZ-PWREL, and ZFP-ACC modes. We omit ZFP-Rate mode here as we find consistent results across all compression ratios. By altering each mode's error bound, we achieve compression ratios of 50×, 25×, 13×, and 7× for each mode. We only show the CESM dataset results as we find similar results with the other datasets.

In Figure 4, the top row corresponds to SZ-ABS, the middle to SZ-PWREL, and the bottom to ZFP-ACC. Each row of graphs displays similar trends with higher compression ratios handling soft errors better, leading to lower percentages of incorrect elements. We achieve these compression ratios using less strict error bounds, which tolerate more error in the data and mask small soft errors in the data as compression errors. For example, to achieve compression ratios of 50× or 25× with ZFP-ACC, we use error bounds of 10 and 0.5, respectively. While the average value in the CESM dataset is 0.3298, these error bounds introduce too much distortion in the data to be used in a real-world context. Therefore, while corruption seems less likely with high compression ratios, the bounds that mask these errors are ill-suited for scientific data.

From these results, we also find artifacts of the compression process. As the compression ratio reduces to 13× and 7×, all graphs exhibit a similar downward slope, with soft errors in bits closer to the compressed data's start causing the most corruption.



(a) CR: 50× (b) CR: 25× (c) CR: 13× (d) CR: 7×

Figure 4: CESM: Percent of elements which violate the set error bound per fault location at increasing levels of loss normalized by compression ratios (CR).
Top: SZ-ABS, Middle: SZ-PWREL, Bottom: ZFP-ACC.

SZ's compression algorithm uses multiple steps to predict and efficiently encode the data using a Huffman tree to reduce the number of bits required for each data value. As a final pass, ZStd further compresses this data. ZStd starts with a dictionary matching stage to further reduce the data before performing finite-state entropy encoding and Huffman encoding. These figures show the structure of ZStd's encoding process, with elements used during the reconstruction process more often towards the start of the compressed data and the less used elements towards the end.

Conversely, ZFP first splits the data into equal-sized blocks, aligns them, and converts them to signed integers. ZFP then uses near orthogonal block transformations to decorrelate block values before grouping the data by leading zeros, truncating it, and encoding each bit-plane one by one. These figures show the effects of this grouping and encoding step with a similar pattern to SZ emerging.

4.5 Error Effects on Data Integrity

As the final step in assessing error effects in lossy compressed data, we quantify the drop in data integrity due to a soft error. To quantify this, we record the decompression bandwidth and compute the maximum absolute difference and PSNR for all *Completed* trials. Figure 5 shows the averages and variances for each of these metrics for each configuration and their corresponding control cases.

Decompression Bandwidth: When examining the gathered decompression bandwidths, we find that all corrupt trials' average bandwidth is close to the control trials. In contrast, the standard deviation of all corrupt trials is higher than the equivalent control trial. These variances are due to unexpected changes in data values and cause more unstable decompression bandwidths.

In rare instances, we find very high and low decompression bandwidths. These are due to corruptions in metadata that control the decompression process, ending the process early or making the process loop near infinitely. This drastic change in decompression bandwidth is an indication of corruption in the data.

Full Dataset Metrics: To investigate the resulting data's integrity after an error, we analyze each trial's maximum absolute difference and PSNR rating. We find the average maximum difference in all trials greatly exceeds the set error bound and is on the order of 10^{19} – 10^{38}. This orders-of-magnitude shift is due to soft errors in bits used to rebuild the more significant bits of data values. The largest shifts occur when the error is in bits used to reconstruct the exponents of data values. However, looking at the variance, these orders-of-magnitude shifts are not always the case. Reviewing the PSNR rating of all trials, we find corrupted trials have significant drops in average PSNR in most cases. However, trials using ZFP-Rate mode did not experience such drops due to its block compression approach. By removing dependencies between data blocks, the error cannot propagate, resulting in the retention of a higher PSNR rating. Looking at all trials' variance, we find the PSNR rating is not easily predicted and fluctuates depending on the error location. This is because the PSNR depends on the percent of incorrect elements and the magnitude shifts experienced.

4.6 Error Effect Observations

With our results, we gain a more in-depth knowledge of how soft errors affect lossy compressed data. First, we find that all tested

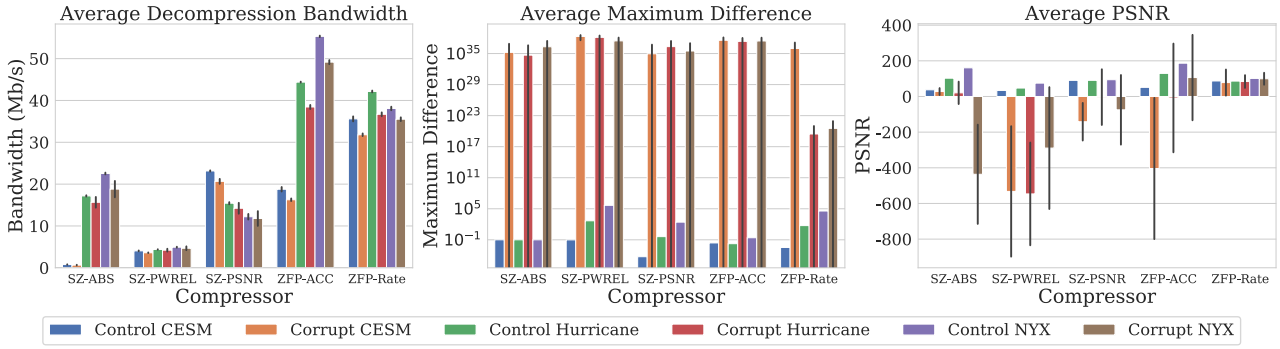


Figure 5: Average data integrity metrics for all trials.

error-bounding modes, excluding ZFP-Rate mode, cannot sufficiently protect against soft errors. Specifically, 95.28% of all trials did not recognize the data's error during the decompression process, leading to error propagation and SDC in later calculations. We also find the location of an error impacts the percent of incorrect elements that occur. However, as the more error sensitive locations depend on a range of factors, all bits require equal protection. Next, we find similar trends across compressors and error bounding modes when varying the compression ratio. These similarities are due to the final encoding stages found within both SZ and ZFP. When analyzing the resulting decompression bandwidth, we find it is not affected heavily by the error in most cases. However, when finding a very high or low bandwidth, this signals possible data corruption. Finally, when examining the resulting data integrity, we find the maximum difference and PSNR rating after an error cannot be easily predicted and can vary drastically.

Out of all error bounding modes tested, we confirm ZFP-Rate mode as the most resilient to soft errors. This is due to it using fully decoupled block-based compression, which prevents errors from propagating. However, as we discuss in Section 2.1, its lower compression ratios and its inability to bound lossy compression error are limiting factors to its broader use.

Overall, we find that neither SZ nor ZFP handles soft errors effectively while maintaining high compression levels in their current state. While adjusting the other modes of SZ and ZFP to function in a similar block-compression manner increases their resiliency, doing so reduces the compression ratios they can achieve. However, efforts are necessary as a single error in their current state leads to an average of ~10% of data values becoming incorrect.

5 ARC: AUTOMATED RESILIENCY FOR COMPRESSION

We use the knowledge obtained in Section 4 to combat the error sensitivity exhibited by lossy compressed data. Our results show that most soft errors go undetected during the decompression process. As such, we must ensure no errors are present before decompression occurs. We also must provide equal protection to all bits since it is difficult to determine the most sensitive in a black box approach.

Current standard techniques to protect data include duplicating data, algorithm-based fault tolerance (ABFT), and ECC. ECC is a popular approach as it requires less space than keeping full backups of data and is application and data agnostic, unlike ABFT. However, with so many ECC approaches, choosing the optimal protection scheme is difficult without knowing each implementation's details.

While hardware-based ECC, such as Chipkill, protects application memory, we choose to use a software-based approach for several reasons. First, hardware-based methods are not always available such as in an OpenScience Grid computing situation. Second, in Section 4, we find that lossy compressed data is susceptible to soft errors, and, as this data stays in storage for long durations, the level of protection provided by hardware may be insufficient. Using a software-based approach enables us to ensure lossy compressed data has sufficient protection that is always available.

To alleviate the difficulty of using ECC and the shortcomings of hardware-based solutions, we develop ARC: Automated Resiliency for Compression as a lightweight software-based solution to streamline the protection of lossy compressed data.

5.1 ARC Interface

The first element of ARC's design and its primary point-of-contact with users is the ARC Interface. The user's data must first be in the form of a `uint_8` byte array to use ARC. Most lossy compressors return the compressed data in this form, but by generalizing the input, any `uint_8` byte array can use ARC for protection. Avramenko et al. found that faults in lossless compressed program variables often leads to silent data corruption [1]. By generalizing, ARC is capable of resolving lossless and lossy sensitivities.

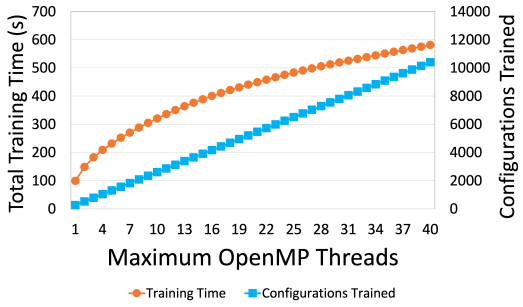


Figure 6: ARC training cost with various numbers of maximum OpenMP threads and the resulting ARC configurations trained.

To use the ARC Interface functions, the user must call ARC's initialization function, `arc_init()`, and provide the maximum number of OpenMP threads available to ARC. OpenMP [2] is a shared-memory multiprocessing programming interface often used to improve the throughput of highly parallelizable tasks. ARC uses OpenMP to apply thread-level parallelism to boost its throughput since different configurations of ARC's ECC approaches do not significantly differ in runtime. Providing the maximum number of OpenMP threads sets a resource utilization cap for ARC and is useful when the host application is fully utilizing system resources, but the user can send `ARC_ANY_THREADS` to remove thread restrictions. With this input, ARC's initialization function loads encoding information resources and begins its configuration training phase.

During this phase, ARC checks its installation directory for a cache of previously saved configurations and loads them if possible. If this is the first time running ARC on a system, ARC trains all ECC configurations on an increasing number of threads up to the maximum available threads and saves the results. If ARC finds only a subset of configurations, it loads the available ones and trains as needed to obtain information on the missing ones. Figure 6 shows the total training time when initializing ARC with various numbers of maximum OpenMP threads on an Intel Xeon 6248G 20 core 2-Way Hyper-Threaded CPU with 372GB of memory. From this graph, we see that as more threads are available, ARC generates more configurations. By having more configurations to choose from, ARC is more capable of meeting the user's constraints. While training extra configurations takes extra time, the overhead increase is logarithmic as ARC uses more OpenMP threads for each training step. This process is only required once for each number of threads. As such, ARC's training phase represents a decreasing amount of ARC's total uptime as it is used more on a system. Once initialized, ARC is ready to encode any `uint_8` byte array using the `arc_encode()` function, which optionally takes a memory constraint, a throughput constraint, and a resiliency constraint.

The memory constraint acts as an upper bound on storage and limits the storage ARC uses when encoding. ARC defines this constraint as the fraction of the byte array size the user wants to add as protection, but the user provides `ARC_ANY_SIZE` to remove this storage restriction. If the user does not want the input size to increase more than 25%, a memory constraint of 0.25 ensures ARC does not exceed this budget. ARC limits its memory overhead through this constraint, enabling it to maintain high data compression ratios.

The throughput constraint acts as a lower bound for the encoding throughput and is defined in units of MB/s, but the user provides `ARC_ANY_BW` to remove this throughput restriction. If the user requires the encoding process to maintain a throughput of 200 MB/s, a value of 200 ensures ARC uses the necessary OpenMP threads to maintain this throughput. As throughput varies on different machines, the training phase's results help parameterize this model. ARC limits its temporal overhead through this constraint, allowing the application to retain close to its original performance.

The resiliency constraint acts as a filter by limiting potential ECC methods to those chosen by the user. This constraint takes an array of any combination of ECC method flag values (`ARC_PARITY`, `ARC_HAMMING`, `ARC_SECDED`, and `ARC_RS`), error-response flag values (`ARC_DET_SPARSE`, `ARC_COR_SPARSE`, and `ARC_COR_BURST`), or the number of expected uniformly distributed soft errors per MB of data. When using the ECC method flags, ARC uses only the specified ECC methods. When using the error-response flags, ARC uses only the ECC methods capable of detecting sparse uniformly distributed errors, correcting sparse uniformly distributed errors, and correcting densely packed burst errors, respectively. Lastly, when directly entering the number of expected uniformly distributed soft errors per MB of data, ARC uses only the ECC methods capable of correcting those errors. For instance, if the user predicts over a sixteenth of each MB of data will encounter a soft error, ARC only uses Reed-Solomon due to the high possibility of burst errors. However, at lower rates, ARC uses SEC-DED or Reed-Solomon as the likelihood of burst errors occurring is lower. Through this constraint, ARC guarantees to provide the desired level of resiliency.

Using these constraints, ARC encodes the data as we show in Figure 7a. ARC begins by passing these constraints to the encoding optimization functions. It then uses the resiliency constraint to filter the potential ECC configurations down to the user-specified set. ARC then looks through this set of configurations to find the one whose memory overhead is under but closest to the memory constraint and whose throughput is above but closest to the throughput constraint. If no configuration meets these requirements, ARC used the ECC method with the memory overhead closest to the memory constraint and the configuration of this method whose throughput is closest to the throughput constraint. Using this configuration, the ARC Engine then encodes the `uint_8` byte array and passes the encoded data back to the user through the ARC Interface.

To access the encoded data, users call ARC's decode function, `arc_decode()`, as we see in Figure 7b. In this case, the ARC Interface passes the encoded data to the correct ARC Engine decoding function, which checks for data errors. If errors are detected and cannot be corrected, ARC sends an error to the user. When ARC

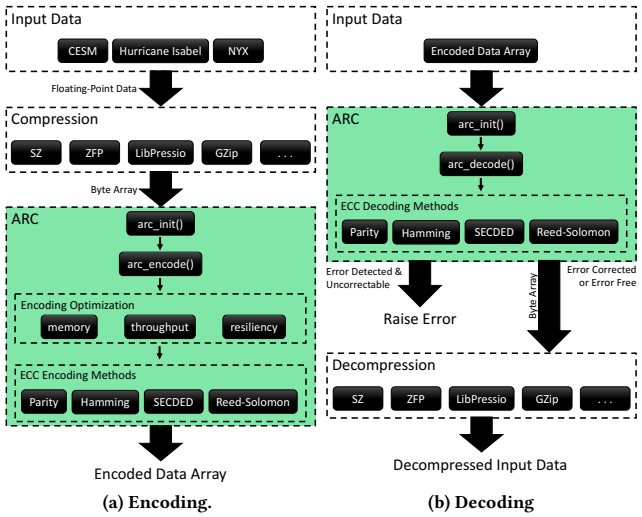


Figure 7: Overview of ARC.

[2] https://www.openmp.org/

finds no errors or the errors are correctable, the byte array is repaired, if necessary, and returned to the user.

When ARC is no longer needed, users must call the `arc_close()` function. This function calls the `arc_save()` function to update all cached configurations with up-to-date versions gathered during normal ARC operations. This step is critical as ARC's throughput may fluctuate, and this ensures all estimations are kept accurate. Finally, this function frees all memory ARC is currently using.

5.2 ARC Engine

Another way to interact with ARC is through the ARC Engine functions we list in Table 1. These functions are split into two main categories: constraint optimization and encode/decode functions.

The constraint optimization functions consist of three functions: `arc_memory_optimizer()`, `arc_throughput_optimizer()`, and `arc_joint_optimizer()`. Each of these accepts a resiliency constraint as well as a memory constraint, throughput constraint, or both. These functions provide ARC's suggestions on which configuration best suits the provided constraints. However, the user can ignore these suggestions for any reason.

The encode/decode functions consist of four pairs of functions, with each pair corresponding to an ECC approach we discuss in Section 2.2. ARC's first ECC is single-bit even parity, which applies a minimal amount of redundancy to each equally-sized data block in the form of a single parity bit to reduce the possibility of SDC. To use this ECC directly, the user must provide the number of data bytes per parity bit. While this approach introduces the least overhead, it is vulnerable to multi-bit errors. The second and third approaches offered are standard Hamming and SEC-DED codes. Both generate parity bits for one byte or eight byte data blocks at a time. While both algorithms have more overhead than parity, they both detect and correct single-bit corruptions, with SEC-DED able to detect 2-bit errors. The final ECC ARC offers is Reed-Solomon encoding by leveraging the abilities of Jerasure[3]. Jerasure is an erasure coding library that efficiently encodes data with various erasure coding algorithms. While Reed-Solomon encoding provides the highest protection of all algorithms, it does result in the most storage overhead and slowest throughput. By offering these ECC algorithms, the ARC Engine offers a full range of protection levels.

Using the ARC Engine directly gives users more control over how their data is protected and enables developers to integrate ARC Engine functions directly into their applications. By integrating ARC as the last step in a lossy compression algorithm or library such as LibPressio, lossy compressed data is vulnerable for a shorter period. However, ARC is not restricted to these ECC or constraint and supports further custom ECC algorithms and constraints to support user needs.

[3]http://web.eecs.utk.edu/~jplank/plank/www/software.html

Functions	
arc_memory_optimizer()	arc_hamming_decode()
arc_throughput_optimizer()	arc_secded_encode()
arc_joint_optimizer()	arc_secded_decode()
arc_parity_encode()	arc_reed_solomon_encode()
arc_parity_decode()	arc_reed_solomon_decode()
arc_hamming_encode()	

Table 1: Available ARC Engine functions.

6 ARC EVALUATION

When developing ARC, we set four main goals for its design: scalability, performance, resiliency, and ease of use. To evaluate ARC, we examine how it scales with extra resources, meets user needs, provides protection, and we demonstrate ARC's ease of use on two recently decommissioned real-world HPC systems.

6.1 Scalability Evaluation

As ARC intends to solve HPC system soft error issues, it is critical to scale efficiently so the host application maintains performance. Therefore, to evaluate ARC's scaling capabilities, we must understand how each ECC algorithm scales with more OpenMP threads.

When evaluating how ARC's provided ECC algorithms scale, we run on an Intel Xeon 6248G 20 core 2-Way Hyper-Threaded CPU with 372GB of memory and set the maximum number of threads for ARC to 40. We run ten trials using the CESM dataset for each ECC and thread count combination with the averages and standard deviation for the encoding and decoding processes and show the results in Figure 8 and Figure 9. From Figure 8, we see nearly all encoding processes scale nearly linearly with more threads. We find the throughput of all methods ranges from 0.04 – 3730 MB/s. When checking the 40 to 1 thread speedup, we find Parity, Hamming, SEC-DED, and Reed-Solomon demonstrate speedups of 19.7×, 26.8×, 33.9×, and 16.4×, respectively. We find similar results in Figure 9 when looking at the decoding processes. In this case, we find the throughput of all methods ranges from 10.64 – 3602 MB/s. When looking at the 40 to 1 thread speedup, we find Parity, Hamming, SEC-DED, and Reed-Solomon demonstrate speedups of 18.6×, 33.5×, 33.5×, and 18.3×, respectively. For both encoding and decoding, we find a wide range of throughputs available to ARC as each ECC algorithm scales differently in parallel. Upon comparing ARC's range of throughputs to SZ's and ZFP's throughputs, we find ARC is more than capable of keeping pace with the less than 200 MB/s SZ and ZFP demonstrate [19]. However, as soft errors require more computations in the decoding process, we must also ensure the decoding process scales when soft errors are present.

To evaluate how soft errors affect the decoding process, we perform two experiments. First, we inject a single correctable soft error into the encoded blocks of data. Second, we inject 100,000 random correctable soft errors into the encoded blocks of data. When injecting the soft errors, we randomly inject the soft errors into the encoded data but also ensure the soft errors are correctable. We do this to evaluate how the correction process affects the throughput. We also do not test single-bit parity as it is only capable of throwing errors and cannot correct the soft errors injected.

Figure 10 demonstrates the resulting throughput when a single and 100,000 correctable soft errors are present in the encoded data. From these graphs, when a single correctable error occurs, the only algorithm's throughput that changes is Reed-Solomon. This is due to Reed-Solomon's high repair costs, which drops the speedup at 40 threads from 18.3× to 2.7×. Comparing the single soft error scenario to the worst-case scenario in which 100,000 correctable soft errors are present, we see significant drops in the throughput

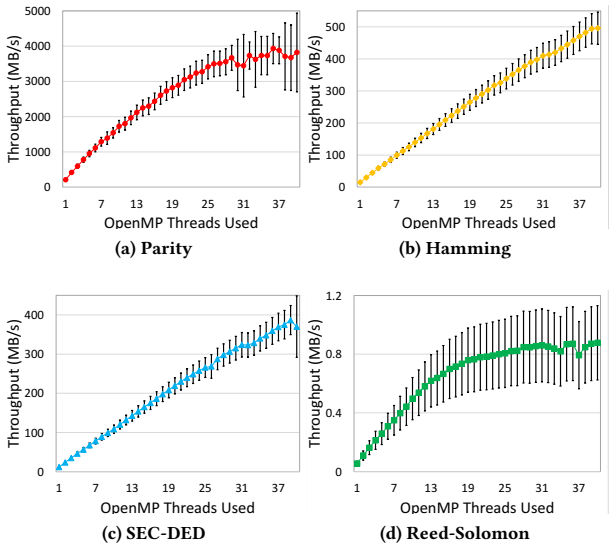


Figure 8: ARC's encoding scalability.

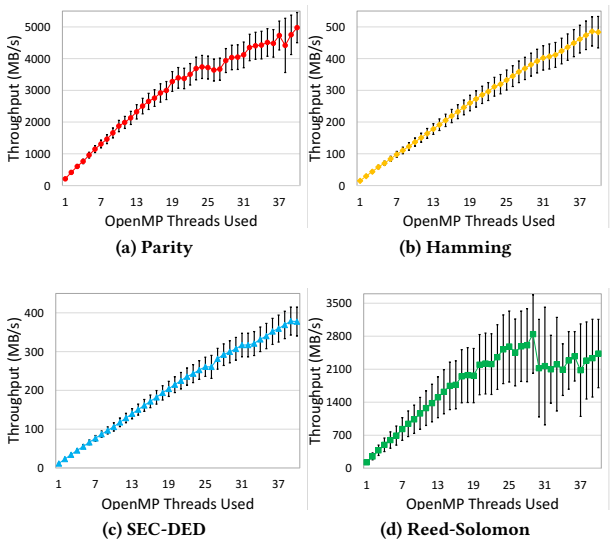


Figure 9: ARC's decoding scalability.

for all three ECC methods. Specifically, the 40 thread speedup of standard Hamming, SEC-DED, and Reed-Solomon drop to 2.64×, 2.43×, and 1.1×, respectively. However, even with these large drops, all three maintain a workable throughput above 7 MB/s and correct the data. Therefore, more correctable errors drop the decoding throughput, but it takes many more errors than would likely occur to reduce the decoding throughput to such low levels. In the more likely case a single correctable soft error occurs, ARC's decoding process maintains its original speedup, with Reed-Solomon being the only exception, which drops 6.7× due to its higher repair costs.

6.2 Performance Evaluation

As each user's needs are unique, one of the main goals in developing ARC is to satisfy user needs efficiently. Therefore, to assess ARC's success, we must evaluate ARC's ability to satisfy user constraints on storage and throughput with any ECC and limited ECC options.

To evaluate ARC, we use the CESM dataset using SZ-ABS and an error bound of $\epsilon = 0.1$ with other datasets yielding similar results. We set the maximum number of threads to 40 and use the same system as in Section 6.1. Figure 11 shows the results of our evaluation when ARC is free to use any ECC method, while Figure 12 shows the results of our evaluation when limiting ARC with the resiliency constraint to use any single ECC method.

Figure 11(a) shows ARC's performance when not limited. In this case, ARC manages to apply the configuration that utilizes the provided space best given the upper bound on storage. For example, when the user does not want the input data's size to increase more than 20% and provides a memory constraint of 0.2, ARC uses Reed-Solomon encoding with 15 code devices over every 241 data devices, resulting in a memory overhead of 19.5%. When given a higher memory constraint of 0.9, ARC uses the extra storage to add increased protection in the form of 103 code devices over every 153 data devices, leading to a memory overhead of 88.5%.

Figure 12(a) shows ARC's memory performance when using the resiliency constraint to limit ARC to any single ECC method. Here, ARC manages to apply the chosen ECC method's configuration that uses the provided space best given the upper bound on storage. However, while limiting ARC to a subset of ECC methods guarantees a protection level, it also reduces ARC's ECC options, and as a result, it cannot always use the entire storage budget. For instance, both Hamming and SEC-DED only have two configurations as stated in Section 5.2 and cannot always use the entire storage budget. Similarly, Parity also develops a step-like function as we only apply single-bit parity on a byte level. ARC also may need to go over the memory constraint when given a low memory constraint and a single ECC method due to its limited options. For instance, a memory constraint of 0.05 and a resiliency constraint that requires Reed-Solomon forces ARC to go over budget, display a warning, and use the Reed-Solomon configuration that results in the lowest memory overhead possible. Overall, ARC uses as much of the storage budget as possible while also avoiding going over budget as much as possible.

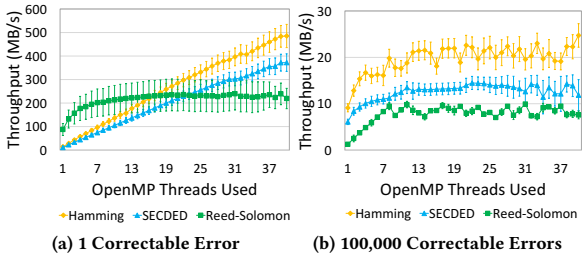
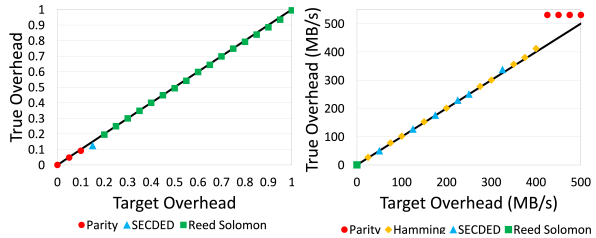


Figure 10: Effect of 1 correctable error and 100,000 correctable errors on ARC's decoding throughput.

Conversely, Figure 11(b) shows the results of using various ARC throughput constraints when ARC is not limited. This graph shows that ARC manages to apply the best configuration with the appropriate number of OpenMP threads to maintain a desired throughput. For instance, when given a throughput constraint of 0.5 MB/s, ARC uses Reed-Solomon over 15 OpenMP threads, resulting in a throughput of 0.51 MB/s. However, with a throughput constraint of 300 MB/s, ARC cannot use the slow Reed-Solomon and instead uses SEC-DED over 34 OpenMP threads, resulting in a throughput of 302.4 MB/s. While ARC could always run each configuration with the maximum number of threads, using fewer threads reduces ARC's impact on available resources and is useful when resources are highly contested in an application.

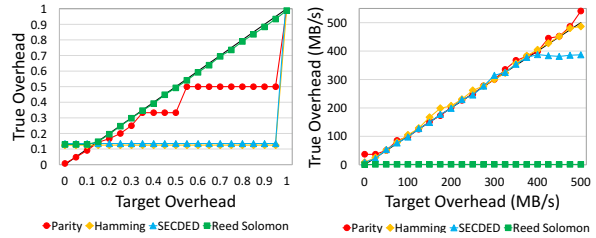
Figure 12(b) shows ARC's throughput performance when using the resiliency constraint to limit ARC to any single ECC method. Here, ARC applies the chosen ECC method's configuration that best meets the lower bound on throughput. Again, while using the resiliency constraint does ensure a protection level, it also reduces ARC's options, and as such, it cannot always meet the necessary throughput requirements. For instance, our Reed-Solomon method is not fast and cannot meet most high throughput requirements. However, if meeting the throughput requirement is impossible for the chosen ECC, ARC attempts to get as close as possible.

ARC also supports using all three constraints in conjunction with one another. When using all three, ARC first uses the resiliency constraint to determine potential ECC configurations. At this point, the memory constraint and throughput constraint will either synergize well or conflict with one another. When these two constraints work well together, and ARC is free to use any ECC methods, we find results similar to those found in Figure 11(a) and (b). For example, when given a memory constraint of 0.2 and a throughput constraint of 0.6 MB/s, ARC uses Reed-Solomon as both other constraints make this possible. However, when the constraints conflict with one another, ARC must make trade-offs to satisfy both constraints. For instance, if given a higher memory constraint of 1 and a higher throughput requirement of 100 MB/s, ARC uses SEC-DED instead since Reed-Solomon cannot achieve the required throughput. This trade-off process is further complicated when ARC's resiliency constraint is more restrictive. When this occurs, even when the memory and throughput constraints agree on an ECC configuration, the resiliency constraint may not allow it. In this case, ARC uses the ECC configuration from the potential ones that best uses the available resources. ARC still satisfies conflicting constraints to the best of its ability when using all constraints together but does not apply the highest level of protection.



(a) Memory Constraint (b) Throughput Constraint

Figure 11: Performance evaluation of `ARC_ANY_ECC`: target memory overhead vs. observed memory overhead.



(a) Memory Constraint (b) Throughput Constraint

Figure 12: Single ECC ARC Performance Evaluation: Target Overhead vs. True Overhead.

6.3 Resiliency Evaluation

To evaluate ARC's ability to provide protection, we apply ARC to each of the previous datasets with a resiliency constraint of 1 error per MB and rerun our fault injection study trials (Section 4). Using this configuration, ARC applies SEC-DED to every eight bytes of data which guarantees to catch any single errors. Upon running these trials, we find that ARC corrects all soft errors we inject. We expect this outcome as ARC uses SEC-DED, but all ARC's provided ECC approaches prevent single-bit soft errors. However, these trials only focus on single-bit errors, which is not always the case.

Even though single-bit errors are the most common form of soft error, ARC must protect from multi-bit errors too. To improve the resiliency ARC provides, users can directly choose the desired level of protection using the resiliency constraint. To further optimize this protection, users can provide a higher storage budget and lower throughput requirements, allowing ARC to use stronger versions of the desired level of protection. For example, when using `ARC_RS` and providing a memory constraint of 0.2, ARC uses a Reed-Solomon configuration with 15 code devices. However, by increasing the memory constraint to 0.9, ARC uses a Reed-Solomon configuration with 103 code devices. While either configuration allows ARC to correct multi-bit and burst errors, the second configuration can correct more corrupt data than the first. Conversely, if the user provides ARC a memory constraint of 0.1, a throughput constraint of 700 MB/s, and does not specify a protection level, ARC uses single-bit even parity. With this ECC approach, ARC only detects single-bit and odd-numbered soft errors within each block of data. Therefore, while ARC can provide sufficient resiliency, it is up to the user to ensure ARC has the proper inputs to protect the data.

6.4 Ease of Use Evaluation

To evaluate how easy ARC is to use on HPC systems, we demonstrate the necessary code changes to integrate ARC and describe how to use ARC on the two recently decommissioned production-level HPC systems discussed in the study by Sridharan et al [31, 33].

To deploy ARC within an application, we must make the correct code changes to integrate ARC into the application. We design ARC so that this process is as minimal as possible, with Algorithm 1

showing the four necessary lines of code. With ARC integrated into the code, we must determine what memory, throughput, and resiliency constraints to provide ARC.

In Algorithm 1, `ARC_ANY_MEM`, `ARC_ANY_BW`, and `ARC_ANY_ECC` represent the memory, throughput, and resiliency constraints. While ARC works with these values and provide the most robust ECC configuration, users may need to change these to satisfy specific needs. However, when choosing constraints, we recommend considering the failure rate of the system as well.

To demonstrate how the failure rate affects the constraint choice, we use the two systems discussed in the study by Sridharan et al. [31, 33]. This study's first system is Cielo, an 8,500 node supercomputer in Los Alamos, New Mexico, situated at around 7,300 feet in elevation. The second system is Hopper and is a 6,000 node supercomputer in Oakland, California, situated at 43 feet in elevation.

Within this study, the authors analyze the failure rate per DRAM device over 30 days for each system. From their findings, the authors find Cielo has nearly twice the failure rate as Hopper. The authors attribute this difference primarily to the difference in altitude of the two systems. Using each of these failure rates and the number of compute nodes on each system, we compute the mean time between failure (MTBF) for each system. We determine Cielo has a failure due to a soft error every 1.9 days, and Hopper ever 5.43 days. However, it is essential to note that this rate does not include undetected soft errors, which cause SDC. Moreover, Section 4 shows a single undetected soft error can lead to unacceptable deviations in lossy compressed data. Thus, decreasing the time between failures.

In their papers, Sridharan et al. also provide a breakdown of all faults they find. On Cielo, they find soft errors caused 34.9% of all faults, and on Hopper, they found soft errors caused 42.1% of all faults. Using this information, we find that single-bit errors caused 70.79% of Cielo's faults, while we find single-bit errors caused 94.6% of Hopper's faults. Using both the failure rate and the distribution of what caused these faults, we are better equipped to choose appropriate ARC constraints.

When running an application on Cielo, applications require more robust protection due to this machine's high failure rate and the lower probability that a single-bit was the cause. Specifically, 29.21% of faults were not from single-bit soft errors and instead were multi-bit errors. By breaking these multi-bit errors down further, we find most occur as burst errors in the same DRAM device. Therefore, applications need the higher level of protection that is provided by the Reed-Solomon algorithm. To ensure ARC uses this algorithm, the user can use the `ARC_COR_BURST` flag, the `ARC_RS` flag, or provide a higher memory constraint and lower throughput constraints, as discussed in Section 5. Alternatively, the user could also use the ARC Engine to choose the ECC configuration manually. By using one of the flags and relaxing the constraints further, ARC provides further protection by using even stronger Reed-Solomon configurations, as discussed in Section 6.3.

Algorithm 1 Integrating ARC

```
Input data Input uint8_t array
Input data_size Size of uint8_t array

arc_init(ARC_ANY_THREADS);

int err = arc_encode(data, data_size, ARC_ANY_MEM,
ARC_ANY_BW, [ARC_ANY_ECC], 1, encoded, encoded_size);
...
err = arc_decode(encoded, encoded_size, decoded, decoded_size);

arc_close();
```

On the other hand, an application running on Hopper does not require as robust of ECC that one running on Cielo needs. Hopper has a much lower failure rate than Cielo, and single-bit flips cause over 90% of soft errors. However, upon breaking down the few multi-bit soft errors found on Hopper, we find 4.05% of these occurred as burst errors and are spatially close to one another. With this information in mind, more robust ECC algorithms, such as Reed-Solomon, are unnecessary in most cases on this system. Therefore, discussing with a system admin and entering the predicted number of errors per MB or using the ECC method they recommend for the system provides adequate protection when using ARC. Following this, users can use the memory and throughput constraints to tune ARC to their specific needs. For example, if the user decides they not only want to detect but correct all single-bit errors, they can enter the predicted number of errors per MB. Alternatively, they can use the `ARC_COR_SPARSE`, `ARC_HAMMING`, or `ARC_SECDED` flags. Either approach ensures no interruptions from single-bit errors.

Our evaluation shows how integrating ARC into any application working with sensitive lossy compressed data requires only four lines of code. We also demonstrate that while ARC allows users to provide ideal constraints, users should consider the system's failure rate and the distribution of faults when setting constraints. Using this knowledge, any user can deploy ARC within their application.

7 CONCLUSIONS AND FUTURE WORK

Compression is a powerful solution to mitigating the stress complex HPC applications are putting on the I/O subsystem. However, a single soft error renders compressed data unusable and few have sought to address this critical vulnerability.

In this work, we perform an in-depth fault injection study to understand soft error effects on lossy compressed data. We find none of the tested error-bounding modes handle errors adequately. While the block-based ZFP-Rate mode prevents error propagation, its lower compression ratios limit its effectiveness. We find a single soft error propagates to on average ~10% of the decompressed data when using the other three modes.

We develop ARC using the findings from our fault injection study. ARC automatically determines the optimal ECC configuration given user constraints on storage, throughput, and resiliency before using this configuration to encode the data. Upon evaluating ARC's abilities, we find ARC satisfies user constraints while displaying encoding and decoding throughput up to 3730 MB/s and 3602 MB/s on a 40 core node. We also find that ARC handles multi-bit errors effectively with user-tunable storage and throughput overheads. Lastly, we display its ease of use and how to consider a system failure rate when determining constraints.

In future work, we plan to improve ARC's abilities by adding additional ECC algorithms and improving existing algorithm implementations. We also plan to use other parallelization paradigms, such as MPI or GPUs, to increase ARC's ECC methods' achievable

throughput. Finally, we aim to implement an API to further simplify the addition of custom ECC algorithms and constraints.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. SHF-1910197, SHF-1943114, MRI-#1725573, and NRT-DESE 1633608. This material is also based upon work supported by the U.S. Department of Energy, Office of Science, Office of Workforce Development for Teachers and Scientists, Office of Science Graduate Student Research (SCGSR) program. The SCGSR program is administered by the Oak Ridge Institute for Science and Education (ORISE) for the DOE. ORISE is managed by ORAU under contract number DE-SC0014664. All opinions expressed in this paper are the authors and do not necessarily reflect the policies and views of DOE, ORAU, or ORISE. We thank the reviewers and the shepherd for their guidance in improving this work. Clemson University is acknowledged for generous allotment of compute time on Palmetto cluster.

REFERENCES

[1] Serhiy Avramenko, Matteo Sonza Reorda, Massimo Violante, and Görschwin Fey. 2017. A high-level approach to analyze the effects of soft errors on lossless compression algorithms. *Journal of Electronic Testing* 33, 1 (2017), 53–64.

[2] Tommaso Benacchio, Luca Bonaventura, Mirco Altenbernd, Chris D Cantwell, Peter D Düben, Mike Gillard, Luc Giraud, Dominik Göddeke, Erwan Raffin, Keita Teranishi, et al. [n.d.]. Resilience and fault-tolerance in high-performance computing for numerical weather and climate prediction. ([n. d.]).

[3] Jon Calhoun, Franck Cappello, Luke N Olson, Marc Snir, and William D Gropp. 2019. Exploring the feasibility of lossy compression for PDE simulations. *The International Journal of High Performance Computing Applications* 33, 2 (2019), 397–410. https://doi.org/10.1177/1094342018762036 arXiv:https://doi.org/10.1177/1094342018762036

[4] Franck Cappello, Sheng Di, Sihuan Li, Xin Liang, Ali Murat Gok, Dingwen Tao, Chun Hong Yoon, Xin-Chuan Wu, Yuri Alexeev, and Frederic T Chong. 2019. Use cases of lossy compression for floating-point data in scientific data sets. *The International Journal of High Performance Computing Applications* 33, 6 (2019), 1201–1220.

[5] Yann Collet and Murray Kucherawy. 2018. Zstandard Compression and the application/zstd Media Type. RFC 8478. https://doi.org/10.17487/RFC8478

[6] Khanh N Dang, Michael Meyer, Yuichi Okuyama, and Abderazek Ben Abdallah. 2016. Reliability assessment and quantitative evaluation of soft-error resilient 3D network-on-chip systems. In *2016 IEEE 25th Asian Test Symposium (ATS)*. IEEE, 161–166.

[7] S. Di and F. Cappello. 2016. Fast Error-Bounded Lossy HPC Data Compression with SZ. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 730–739. https://doi.org/10.1109/IPDPS.2016.11

[8] Catello Di Martino, Zbigniew Kalbarczyk, Ravishankar K Iyer, Fabio Baccanico, Joseph Fullop, and William Kramer. 2014. Lessons learned from the analysis of system failures at petascale: The case of blue waters. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, 610–621.

[9] Bo Fang. 2020. *Approaches for building error resilient applications*. Ph.D. Dissertation. University of British Columbia.

[10] Lisa Fischer, Sebastian Götschel, and Martin Weiser. 2018. Lossy data compression reduces communication time in hybrid time-parallel integrators. *Computing and Visualization in Science* 19, 1 (01 Jun 2018), 19–30. https://doi.org/10.1007/s00791-018-0293-2

[11] J. L. Gailly. 1992. *GZIP*. http://www.gzip.org

[12] Thaylon Guedes, Leonardo A Jesus, Kary ACS Ocaña, Lucia MA Drummond, and Daniel de Oliveira. 2020. Provenance-based fault tolerance technique recommendation for cloud-based scientific workflows: a practical approach. *Cluster Computing* 23, 1 (2020), 123–148.

[13] Andy A Hwang, Ioan A Stefanovici, and Bianca Schroeder. 2012. Cosmic rays don't strike twice: understanding the nature of DRAM errors and the implications for system design. *ACM SIGPLAN Notices* 47, 4 (2012), 111–122.

[14] Ravi Kumar Jain. 1992. *Scheduling data transfers in parallel computers and communications systems*. Ph.D. Dissertation. University of Texas at Austin.

[15] Sihuan Li, Sheng Di, Kai Zhao, Xin Liang, Zizhong Chen, and Franck Cappello. 2020. SDC Resilient Error-bounded Lossy Compressor. arXiv:2010.03144 [cs.DC]

[16] Sihuan Li, Sheng Di, Kai Zhao, Xin Liang, Zizhong Chen, and Franck Cappello. 2020. Towards End-to-end SDC Detection for HPC Applications Equipped with Lossy Compression. In *Proceedings of the 22nd IEEE International Conference on Cluster Computing*. IEEE.

[17] X. Liang, S. Di, D. Tao, S. Li, S. Li, H. Guo, Z. Chen, and F. Cappello. 2018. Error-Controlled Lossy Compression Optimized for High Compression Ratios of Scientific Datasets. In *2018 IEEE International Conference on Big Data (Big Data)*. 438–447.

[18] X. Liang, S. Di, D. Tao, S. Li, B. Nicolae, Z. Chen, and F. Cappello. 2019. Improving Performance of Data Dumping with Lossy Compression for Scientific Simulation. In *2019 IEEE International Conference on Cluster Computing (CLUSTER)*. 1–11.

[19] Xin Liang, Ben Whitney, Jieyang Chen, Lipeng Wan, Qing Liu, Dingwen Tao, James Kress, Dave Pugmire, Matthew Wolf, Norbert Podhorszki, and et al. 2020. MGARD+: Optimizing Multilevel Methods for Error-bounded Scientific Data Reduction. *arXiv:2010.05872 [cs]* (Nov 2020). http://arxiv.org/abs/2010.05872 arXiv: 2010.05872.

[20] P. Lindstrom. 2014. Fixed-Rate Compressed Floating-Point Arrays. *IEEE Transactions on Visualization and Computer Graphics* 20, 12 (Dec 2014), 2674–2683. https://doi.org/10.1109/TVCG.2014.2346458

[21] Joseph Nardi, Noah Feldman, Andrew Poppick, Allison Baker, and Dorit Hammerling. 2018. *Statistical Analysis of Compressed Climate Data*. Technical Report. NCAR.

[22] C. Nguyen and G. R. Redinbo. 2005. Fault tolerance design in JPEG 2000 image compression system. *IEEE Transactions on Dependable and Secure Computing* 2, 1 (2005), 57–75.

[23] Xiang Ni, Tanzima Islam, Kathryn Mohror, Adam Moody, and Laxmikant V Kale. 2014. Lossy compression for checkpointing: Fallible or feasible?. In *Poster Session of the 2014 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC '14)*. IEEE Computer Society, Washington, DC, USA.

[24] John Ousterhout and Fred Douglis. 1989. Beating the I/O Bottleneck: A Case for Log-Structured File Systems. *SIGOPS Oper. Syst. Rev.* 23, 1 (Jan. 1989), 11–28. https://doi.org/10.1145/65762.65765

[25] Mizanur Rahman, Mhafuzul Islam, Jon Calhoun, and Mashrur Chowdhury. 2019. Real-Time Pedestrian Detection Approach with an Efficient Data Communication Bandwidth Strategy. *Transportation Research Record* 2673, 6 (2019), 129–139. https://doi.org/10.1177/0361198119843255 arXiv:https://doi.org/10.1177/0361198119843255

[26] Naoto Sasaki, Kento Sato, Toshio Endo, and Satoshi Matsuoka. 2015. Exploration of Lossy Compression for Application-Level Checkpoint/Restart. In *Proceedings of the 2015 IEEE International Parallel and Distributed Processing Symposium (IPDPS '15)*. IEEE Computer Society, Washington, DC, USA, 914–922. https://doi.org/10.1109/IPDPS.2015.67

[27] Bianca Schroeder, Eduardo Pinheiro, and Wolf-Dietrich Weber. 2011. DRAM errors in the wild: a large-scale field study. *Commun. ACM* 54, 2 (2011), 100–107.

[28] Baodi Shan, Aabid Shamji, Jiannan Tian, Guanpeng Li, and Dingwen Tao. 2020. LCFI: A Fault Injection Tool for Studying Lossy Compression Error Propagation in HPC Programs. arXiv:2010.12746 [cs.DC]

[29] Taniya Siddiqua, Athanasios E Papathanasiou, Arijit Biswas, and Sudhanva Gurumurthi. 2013. Analysis and modeling of memory errors from large-scale field data collection. In *SELSE*.

[30] Seung Woo Son, Zhengzhang Chen, William Hendrix, Ankit Agrawal, Wei keng Liao, and Alok Choudhary. 2014. Data Compression for the Exascale Computing Era - Survey. *Supercomputing frontiers and innovations* 1, 2 (2014). http://superfri.org/superfri/article/view/13

[31] Vilas Sridharan, Nathan DeBardeleben, Sean Blanchard, Kurt B. Ferreira, Jon Stearley, John Shalf, and Sudhanva Gurumurthi. 2015. Memory Errors in Modern Systems: The Good, The Bad, and The Ugly. *SIGARCH Comput. Archit. News* 43, 1 (March 2015), 297–310. https://doi.org/10.1145/2786763.2694348

[32] Vilas Sridharan and Dean Liberty. 2012. A study of DRAM failures in the field. In *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–11.

[33] Vilas Sridharan, Jon Stearley, Nathan DeBardeleben, Sean Blanchard, and Sudhanva Gurumurthi. 2013. Feng shui of supercomputer memory positional effects in DRAM and SRAM faults. In *SC'13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–11.

[34] Li Tan and Nathan DeBardeleben. 2019. Failure Analysis and Quantification for Contemporary and Future Supercomputers. *arXiv preprint arXiv:1911.02118* (2019).

[35] R. Underwood. 2020. *LibPressio*. https://github.com/robertu94/libpressio

[36] Robert Underwood, Sheng Di, Jon C. Calhoun, and Franck Cappello. 2020. FRaZ: A Generic High-Fidelity Fixed-Ratio Lossy Compression Framework for Scientific Floating-point Data. arXiv:2001.06139 [cs.DC]

[37] X. Wei, R. Zhang, Y. Liu, H. Yue, and J. Tan. 2019. Evaluating the Soft Error Resilience of Instructions for GPU Applications. In *2019 IEEE International Conference on Computational Science and Engineering (CSE) and IEEE International Conference on Embedded and Ubiquitous Computing (EUC)*. 459–464.