Resilient Error-Bounded Lossy Compressor for Data Transfer

Sihuan Li
University of California, Riverside
Riverside, CA 92521, USA
sli049@ucr.edu

Sheng Di
Argonne National Laboratory
Lemont, IL 60439, USA
sdi1@anl.gov

Kai Zhao
University of California, Riverside
Riverside, CA 92521, USA
kzhao016@ucr.edu

Xin Liang
Missouri University of Science and Technology
Rolla, MO 65409, USA
xliang@mst.edu

Zizhong Chen
University of California, Riverside
Riverside, CA 92521, USA
chen@cs.ucr.edu

Franck Cappello
Argonne National Laboratory
Lemont, IL 60439, USA
cappello@mcs.anl.gov

ABSTRACT

Today's exa-scale scientific applications or advanced instruments are producing vast volumes of data, which need to be shared/transferred through the network/devices with relatively low bandwidth (e.g., data sharing on WAN or transferring from edge devices to supercomputers). Lossy compression is one of the candidate strategies to address the big data issue. However, little work was done to make it resilient against silent errors, which may happen during the stage of compression or data transferring. In this paper, we propose a resilient error-bounded lossy compressor based on the SZ compression framework. Specifically, we design a new independent-block-wise model that decomposes the entire dataset into many independent sub-blocks to compress. Then, we design and implement a series of error detection/correction strategies elaboratively for each stage of SZ. Our method is arguably the first algorithm-based fault tolerance (ABFT) solution for lossy compression. Our proposed solution incurs negligible execution overhead in the fault-free situation. Upon soft errors happening, it ensures decompressed data strictly bounded within user's requirement with a very limited degradation of compression ratio and low overhead.

CCS CONCEPTS

• Software and its engineering → Software reliability; Software fault tolerance.

KEYWORDS

Lossy compression, Algorithm Based Fault Tolerance, data transfer

ACM Reference Format:
Sihuan Li, Sheng Di, Kai Zhao, Xin Liang, Zizhong Chen, and Franck Cappello. 2021. Resilient Error-Bounded Lossy Compressor for Data Transfer. In The International Conference for High Performance Computing, Networking, Storage and Analysis (SC '21), November 14–19, 2021, St. Louis, MO, USA. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3458817.3476195

Corresponding author: Sheng Di, Mathematics and Computer Science Division, Argonne National Laboratory, 9700 Cass Avenue, Lemont, IL 60439, USA.

SC '21, November 14–19, 2021, St. Louis, MO, USA

ACM ISBN 978-1-4503-8442-1/21/11.
https://doi.org/10.1145/3458817.3476195

1 INTRODUCTION

Processing scientific data on various devices and transferring them through different types of network is becoming very common [32]. On the one hand, numerous domain scientists are sharing the research data due to ever-emerging open non-trivial scientific problems in different domains. However, today's scientific experiments can easily produce extremely large amounts of data. For example, the SKA (Square Kilometer Array) telescope will generate hundreds of PB/year of processed science data products in 2023 and the HL-LHC will generate 1 EB of science data in 2026. Many other experiments project to generate EBs of science data [9]. These vast volumes of data would be accessed and analyzed by many scientists with common interests at different sites. Also large-scale simulation campaigns that are difficult to reproduce are often shared by many researchers in different sites. For example the Community Earth System Model for the Coupled Model Intercomparison Project (CMIP) 5 has produced about 2.5 PB of data. CMIP6 is projected to exceed 10 PB of the raw data requirements [11]. Eficiently transferring the data on wide area network (WAN) is critical to guaranteeing that the data sharing would not hinder the research progress. On the other hand, transferring large amounts of data from edge devices to remote servers or supercomputers is becoming very common in many research areas. For instance, Linac Coherent Light Source (LCLS-II) may produce an extremely large amount of X-ray imaging data (250GB/s [16]), which needs to be transferred to data centers or supercomputers for post hoc analysis. Another example is that the remote sensor technology continues to increase in fidelity for space systems, so large amounts of data are being collected by orbiting satellites or space vehicles and transmitted to other stations (e.g., ground stations, other satellites) [14].

Error-bounded lossy compressors [21, 35, 38, 39, 48] have been effective in significantly reducing large volumes of data produced by scientific simulations [12, 25, 47] or instruments/devices [22, 42]. They have been thought of as one of the best ways to resolve the big data transferring issue. For instance, Globus [23] is arguably the most eficient data transferring platform, while its maximum throughput is about 8GB/s [40] even with a very high concurrency (transferring hundreds of files concurrently), and its transferring

throughput is only several hundreds of megabytes with a low concurrency. Without compression, transferring 10 petabytes of data may take 120 days on a 1GB/s connection.

Silent errors can lead to incorrect logic unit result or corrupt storage systems silently without notice of hardware or operating system. A recent study from Google has revealed that silent errors are more common than anticipated [26]. Silent errors are non-negligible when running lossy compressors on various end-points such as edge devices and supercomputers:

- Silent errors in supercomputers: If one lossy compressor is employed to compress a large amount of data on a supercomputer in parallel, silent errors have to be taken into account. In general, the single-core lossy compression throughput is only 20MB/s∼200MB/s [15, 55] for an error-bounded lossy compressor (e.g., SZ [21, 35, 48], ZFP [38] and MGARD [8]), while vast volumes of data need to be compressed at runtime (e.g., a single exascale execution of cosmological simulation [25] may produce 20PB), so hundreds of thousands of core hours are required for the compression. Silent data corruption (SDC) may occur in the entire compression procedure because of the large amount of resources and time involved (e.g., memory, cache, register, CPU, for hours). The SDCs are mainly referred to as computational silent errors and memory silent errors. Such errors are very dangerous because they may mislead the scientists in research studies.
- Silent errors in error-prone devices/environment: The edge devices especially in the special environments such as interplanetary space probe in orbiting satellites or space vehicles would be more error-prone (mainly struck by SDCs) than the regular devices on the earth. To address this issue, some fault tolerance techniques [30, 37] have been proposed for specific algorithms such as matrix multiplication and FFT. However, when lossy compressors are used by such particular devices to compress image data, the whole compression procedure has to be protected against soft errors. Otherwise, the corrupted data may let scientists miss important findings or draw a misleading conclusion.
- Potential malicious attacks or missing/erroneous packets: Sharing large amount of scientific data between scientists or institutions is becoming very common, so eficiently and reliably transferring such vast volumes of data on wide area network (WAN) is significant to the scientific community. Whenever the data need to be shared on WAN, they will face serious silent error issues such as potential malicious attacks or missing/erroneous packets (e.g., if UDP is used).

Mat Noor and Vladimirova [43] made a parallel fault-tolerant Integer KLT implementation for lossless hyperspectral image compression on board satellites. However, there are no lossy compressors designed to detect and correct silent errors. Designing an silent error detection/correction method for lossy compression is challenging because decompressed data will deviate from original data even though there are no silent errors occurring.

In this paper, we develop a resilient error-bounded lossy compressor based on SZ [35] - one of the best generic error-bounded lossy compressors for large-scale scientific datasets [35, 41]. SZ allows users to control compression errors in different ways (such as absolute error bound, relative error bound and peak signal-to-noise ratios) and it can get very high compression ratios with satisfied reconstruction quality [24, 31, 36, 45, 48], so it has been widely used or evaluated in the scientific community. Not only can our solution detect the possible silent errors during the compression/decompression but it can also automatically correct the errors in many cases. This is very helpful in detecting/correcting the soft errors during the data transfer (such as the errors occurring in the error-prone UDP policy or the ones caused by malicious attacks). Specifically, the independent-block design in our compressor allows to recover the data by retrieving only a small piece of data, which would be very helpful in data transfer on WAN as the data sender/transmitter just needs to resend a tiny amount of data instead of the whole dataset upon any error detection.

The main idea of this paper is analyzing each subroutine in the SZ lossy data compression framework elaborately and designing a series of fault tolerance strategies carefully, such that the lossy compressor can be protected against silent errors effectively with little overhead. We summarize the detailed contributions as follows.

- We comprehensively analyze each subroutine of SZ with respect to possible memory/computation errors. The analysis unveils that some parts of SZ are naturally error resilient, while other parts are fragile to silent errors. Silent errors striking these parts may cause wrong decompressed data or even crash. Thus, it is critical to protect those parts by specific fault tolerance strategies.
- We propose an eficient resilient lossy compression solution based on the SZ compressor. We modify SZ by dividing each dataset into small blocks and making the compression work totally independent across blocks. Such a design is able to control the impact of silent errors on the decompressed data and also fix the data with minimum overhead. We design resilient strategies which can not only detect silent errors in most of cases but also correct them automatically in some cases.
- We implement our resilient compressor and evaluate: (i) its fault tolerance ability in the presence of silent errors, (ii) the corresponding overhead in the fault-free situation and (iii) the possible impact to the compression quality. We perform the experiments with real-world simulation data across multiple science domains and image data taken by New Horizons probe [4] in aerospace. Experiments show that our designed independent-block based compression model has very limited execution overheads (≤10% in most cases). The experiments also confirm that our fault tolerance solution yields little overhead (≤7.3% at 2048 cores) and correct decompression results in the presence of soft errors. When injecting one and two errors, respectively, during the compression at runtime, our solution can significantly improve SZ resilience (92% running cases with correct decompressed data compare to only 71.2% and 47% of the original SZ).

We organize our paper as follows. Section 2 discusses related work. Section 3 formulates the research problem. Section 4 provides an in-depth analysis of the fault tolerance ability of SZ. Section 5 presents our fault tolerance methodology. We evaluate our methods in Section 6. The last section concludes the paper.

2 RELATED WORK

We discuss the related work in two facets: the fault tolerance ability of existing lossy compressors and the existing solutions designed to protect other applications against silent errors.

So far, there have been many lossy compressors [19, 21, 33, 35, 38, 39, 47, 48] developed to significantly reduce the large volume of data produced by scientific simulations. All the lossy compressors, basically, could be classified into two categories - transform-based compression [38, 47] and prediction-based compression [21, 35, 39, 48]. None of the transform-based compressors are immune to the silent errors. In fact, if the data in the transformed domain are corrupted because of memory or computation error, multiple data values in the original data domain could be affected. As for the prediction-based model, the silent errors could be also fatal to the reconstruction of data. In SZ, for example, if the data prediction on some data point is struck silently during the compression, the predicted value on that data point would be inconsistent during the compression and decompression, leading to uncontrolled decompression errors.

Much work has been done to fight against memory and computation errors. At the hardware level, error correcting code (ECC) detects and corrects bit flips in memory. ECC can correct single-bit flipped memory errors but cannot detect or correct any computation errors. Hardware redundancy adopts redundant hardware to execute the same application with the same input and compare the outputs from the different hardwares. Software redundancy means running the same program on the single hardware multiple times and compare the outputs from different runs. Thus, double modular redundancy (DMR) is needed for error detection with 100% overhead and triple modular redundancy (TMR) is needed for error correction with 200% overhead.

Such high overhead of modular redundancy to handle soft errors has motivated algorithm based fault tolerance (ABFT) [27], which aims to exploit the special characteristics of an application or algorithm to detect and correct soft errors. Despite the fact that ABFT requires a significant algorithm integration effort, the tiny overhead of ABFT makes it very attractive. Most of the existing ABFT methods, however, focus on popular arithmetic algorithms such as matrix operations [18, 27, 54], fast Fourier transforms [34] and iterative methods [49]. To the best of our knowledge, no ABFT work has been done for lossy compression algorithms, which is a significant gap in the context of scientific data compression.

3 BACKGROUND AND PROBLEM FORMULATION

We discuss the research background and formulate the research problem in this section.

3.1 SZ Lossy Compression Framework

SZ [35] is an error-bounded lossy compressor designed for scientific data. According to the recent studies [35, 41, 48], it can effectively reduce the data size for many scientific simulations, such as climate simulation, cosmological simulation, quantum simulation, and chemical simulation.

Basically, SZ includes four critical stages during the compression: (1) data prediction, (2) linear-scaling quantization, (3) variable-length encoding, and (4) lossless compression such as Zstd [7]. In the data prediction step, SZ [21, 35, 48] splits the whole dataset into multiple blocks and then perform the compression in each block based on two alternative prediction methods - an improved Lorenzo predictor [29] or linear regression. The second step - linear-scaling quantization converts each raw data value (such as floating-point value) to an integer index (or quantization bin) based on the user-set error bound and the difference of the predicted value and original value. The remaining two steps are used to reduce the data size by performing Huffman encoding on the quantization bin index array and adopting lossless compression. This may significantly reduce the data size because the distribution of quantization bin indices are likely fairly non-uniform especially when the data are relatively smooth in space.

3.2 Algorithm based Fault Tolerance (ABFT)

ABFT achieves silent error detection and correction by leveraging the characteristics of the algorithms. In high level explanations, ABFT detects errors by checking if some relationship is respected and correct the errors by another introduced set of computation. Each ABFT technique has to be developed for a particular approach composed by one or more algorithms. We give an example to illustrate how ABFT detects/corrects soft errors in general. Given an array $D[]$ at timestamp t_0, then at a later timestamp t_1, one attempts to detect if there was a memory error that corrupted a value in $D[]$ during the period $[t_0, t_1]$. In order to detect the error, we can leverage a checksum (equation 1).

$$c = \sum_{i=0}^{n} D[i] \quad (1)$$

Specifically, we can calculate the sum of $D[]$ at t_0 and t_1, respectively. Suppose the two calculated sums are denoted by c_{t_0} and c_{t_1}, respectively. If $c_{t_0} \neq c_{t_1}$, we can conclude there must be an error happening to $D[]$ during the period $[t_0, t_1]$. In order to locate where the error is in the array $D[]$, we can leverage an extra computation (equation 2).

$$c' = \sum_{i=0}^{n} i \cdot D[i] \quad (2)$$

Specifically, assuming the value at index k is corrupted during the time period $[t_0, t_1]$, according to $c_{t_1} - c_{t_0} = D[k]' - D[k]$ and $c'_{t_1} - c'_{t_0} = k \cdot (D[k]' - D[k])$, one can derive the error location index $k = (c'_{t_1} - c'_{t_0})/(c_{t_1} - c_{t_0})$. This example illustrates that it is viable to detect and even correct the single-data-point error just by introducing a few more light-weight computations.

3.3 Error Model and Assumptions

We identify the error model in this subsection. In our study, we focus on different types of errors (mainly memory error and computation error). As for the memory error model, the errors could randomly happen anywhere in the whole memory at any time during the life time of a process in the form of bit-flips. As for the computation errors, their impact could appear in the form of bit-flips on the computation results. Similar to other ABFT research, the flow control error (FCE) is beyond the scope of our work because the general solutions are designed on the compiler/instruction/hardware level [46]. Moreover, it is too dificult to comprehensively detect the FCEs even for professional FCE detection tools according to recent

studies [46]. Without loss of generality, we assume that the occurrence probability of multiple computation errors or memory errors is extremely low for one block of data during one compression, since one block is generally very small (such as 10×10×10 in size). Similar to other ABFTs [17, 34, 53], we assume the checksum itself is error free because of its tiny computation time compared with the compression time.

3.4 Formulation of Silent Error Detection Evaluation in SZ

As mentioned previously, SZ has four stages in the whole course of compression, and we mainly focus on the single-data-point silent error (either computation error or memory error) happening at each stage, without loss of generality. In addition, we mainly focus on the dominant data structures (i.e., all the data structures taking linear space of the number of data points □) that take the majority of memory footprint in SZ because they are the major objects affected by silent errors if any. The rest parts (called negligible space in the following text) could be considered error free. Which parts taking negligible space will be discussed later in this paper.

The objective of our work is to detect and correct both computational errors and memory errors in each stage of SZ compression as much as possible. There are three important metrics to evaluate our designed resilient lossy compressor, as listed below.

- Silent error detection/correction ability. What kinds of silent errors could be detected or corrected? What is the accuracy and coverage rate of silent error detection?
- Computational Overhead. It is the ratio of the extra time to the total original execution time in an error-free situation.
- Impact to Compression Result. Whether the resilient lossy compressor can still respect the user-specified error bound for the decompressed data? What is the compression overhead: i.e, how much the compression ratio would be degraded under the resilient compressor compared with the original compressor?

All the three evaluation metrics can be used to all lossy compressors, which is the first resilience formulation in the context of lossy compression, to the best of our knowledge.

4 RESILIENCE ANALYSIS OF SZ 2.1

In this section, we analyze the resilience (the ability of detecting/correcting silent errors and the impact for the undetected errors) for the latest version of SZ - SZ 2.1 based on its principle.

4.1 Resilience Regarding Computation Error

We analyze SZ's natural resilience based on when/where the computation error could happen, including calculation of regression coeficients, selecting bestfit predictor by sampling method, and data prediction and calculation of decompressed data, huffman encoding and lossless compression. We call the first two stages "prediction preparation".

4.1.1 Resilience in the prediction preparation. A computation error in prediction preparation stage may only lower compression ratio to a certain extent but it would not affect the correctness of decompressed data (i.e., still strictly respecting error bound). That is, the decompressed data is still the golden result in spite of the computation error in prediction preparation. In fact, although the computation error may lead to inaccurate regression coeficients or incorrect bestfit predictor selection, exactly the same coeficients/selection will be used for both compression and decompression. The compression ratio could be affected because the data prediction may be less accurate due to the inaccurate coeficients or incorrect predictor selection.

4.1.2 Resilience in the data prediction and calculation of decompressed data. Data prediction is the most critical step in SZ. In order to guarantee the error bound, the neighboring data values used to predict each data point during the compression have to be exactly the same values to be used during the decompression. That is, SZ needs to obtain the decompressed data values during compression. We demonstrate the key compression procedure in the Algorithm 1, which is conducted in a loop of scanning all data blocks.

Algorithm 1 Key Compression Step Per Block in SZ2.1

```
Input: One block of raw data
 1: for (each data point (denoted ori) in a block) do
 2:     pred ← P(neighbor decompressed data);
 3:     Δ ← ori−pred;
 4:     bin ← |Δ|/□ /*Compute quantization bin based on error bound □*/
 5:     if (bin < bin_max) then
 6:         decmp ← □(pred, bin, □); /*Retrieve the decompressed data*/
 7:         if (|ori − decmp| > □) then
 8:             Do unpredictable data compression; /*IEEE 754 binary compression*/
 9:         end if
10:     else
11:         Do unpredictable data compression; /*IEEE 754 binary compression*/
12:     end if
13: end for
```

The SZ compression pipeline involves 5 key steps.

(1) Calculate predicted value (line 2).
(2) Compute the difference between the real value and the predicted value (line 3).
(3) Calculate error quantization bins (line 4).
(4) Calculate the decompressed data (line 6) which will be used to predict the following data points in compression.
(5) Double-check the correctness of the compression based on the given error bound against possible machine epsilon error (line 7-9): specifically, the decompressed value would be reconstructed based on the quantization bin and compared with the true value.

In the following, we analyze the fault tolerance ability of the key procedure of compression upon a computation error occurring in the code segment presented in Algorithm 1 using Figure 1, based on five possible cases. Figure 1 presents a snippet of data in the whole dataset, where the current data point refers to the data point which is being compressed. The red circle points are the original raw data values, and the blue cross point refers to the predicted data value at the current data point. In SZ2.1, each prediction error will be converted to an integer instance discretized based on the quantization bins, whose size is 2× as large as user's error bound. We note that the resilience issue should be discussed in three situations/types regarding different zones as shown in the figure. More specifically, the necessary condition to obtain correct decompressed output is that a correct decompressed value must be calculated (type-1) or an unpredictable data handling is called (type-2) during compression; and the same data should be reconstructed during decompression (type-3), which will be used later.

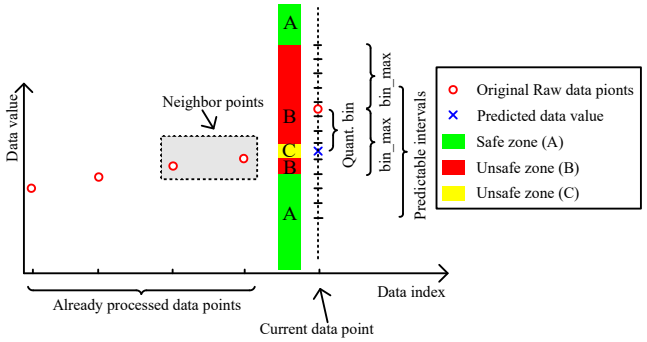


Figure 1: Analysis of fault tolerance ability for SZ with computation error

Case 1 - a computation error happens to line 2. In this case, we need to take into account two possible situations in terms of the deviation of the predicted value affected by the error.

- Situation 1: the predicted value is changed by the error significantly such that the quantization bin calculated later on falls outside the maximum quantization range (i.e., bin < bin_max does not hold). In this situation (zone A in Figure 1), the decompressed data will still respect the error bound for sure because of the type-2 behavior.
- Situation 2: the impact of the silent error on the predicted value is relatively small such that the quantization bin is within the maximum quantization range (i.e., bin < bin_max still holds). This may cause a significant error to the decompressed data (zone B, C in Figure 1) because of violation of type-3 behavior. The reason is described as follows. On the one hand, the double-checking step (line 7) cannot detect such an error because it would decompress the data point based on the "wrong" predicted value such that the reconstructed value will still respect the error bound. On the other hand, it is unlikely that such an silent error would happen again during the decompression, so that SZ would get a different predicted value for the current data point in the course of decompression and thus a wrong decompressed value on this data point (violation of type-3 behavior). Moreover, this decompressed value would also be used to predict other data points in the decompression, so that the errors would be propagated throughout the whole dataset.

Case 2 - A computation error happens to line 3 or 4. These two lines are naturally resilient due to the type-2 behavior. The unpredicatable data compression is always called (line 10 for zone A and line 8 for zone B), no matter how much the calculated quantization bin deviates (zone B or zone A),

Case 3 - A computation error happens to line 6. This may affect correctness of the decompression data, which will be analyzed based on two possible situations.

- Situation 1: the decompressed data value is deviated significantly because of the silent error such that the following double-checking (i.e., line 7-8) suggests to use unpredictable compression here. So it is resilient because of type-2 behavior.
- Situation 2: the decompressed data value is changed slightly such that it skips the double-checking step. In this situation, the skewed (wrong) decompressed data value would be used in the prediction of the succeeding data points, and this would lead to the inconsistent prediction results between the compression and decompression. Thus it is not resilient because of violation of type-3 behavior.

Case 4 - A computation error happens to line 7. Line 7 has very good resilience but not perfect. Obviously, if line 7 makes a false result to be true, it is resilient because of the unpredictable data solution (type-2 behavior). If line 7 makes a true result to be false, it is not resilient because of the impact of machine epsilon. However, in our fault tolerant design, we do not protect this part because the likelihood of this situation is extremely small. This situation happens only when the original real value is located right on the edge of a quantization bin. To be more specific, a test shows only 24 out of 512^3 data points (NYX dataset, relative error bound 1E-3) will make line 7 true.

4.1.3 Resilience in lossless compression.

We will show our solutions are able to detect silent errors that occur in lossless compression in Section 5.3.

All in all, in terms of the SZ lossy compression framework, the only concern regarding fault tolerance during the compression procedure is on the correctness of the predicted value (i.e., line 2 in Figure 1 (a)) and the correctness of data decompression during the compression (i.e., line 6). To address this issue, we adopted an eficient selective instruction duplication method, to be described in Section 5 in detail.

4.2 Resilience Regarding Memory Error

Now, we analyze the resilience against the memory errors occurring in different places, such as input data, regression coeficients and quantization bin index array, respectively.

4.2.1 Resilience against memory error in inputs.

Since the input data (i.e., original data) occupies the significant portion of the memory footprint, we have to protect it against potential silent errors. The input data is used in the following steps: 1. computing the regression coeficients; 2. sampling and estimating the compression error of both regression and Lorenzo predictor; 3. data prediction and calculation of the difference between predicted data and original data and handling unpredictable data. We find that: for the first two steps, similar to the analysis in Section 4.1.1, the memory error in input data will only impact the compression ratio and keep the correctness of decompressed data. However, step 3 must use genuine uncorrected input data since that is where the compression happens. With a corrupted input in step 3, the decompressed data will be calculated based on that corrupted value which is obviously error prone.

We will leverage the above finding to reduce the overhead of checksum calculations since it discloses the fact that the corrupted values may not affect the correctness of decompressed data in the first 2 steps (i.e., error detection/correction for those parts are not necessary).

4.2.2 Resilience against the memory error in regression coefficients.

The memory usage of regression coeficients are found to be very small compared to the overall memory usage such that this part does not need particular protection. Each data block will maintain at most 4 coeficients (for 3D dataset). Thus, the coeficients only take $\frac{4}{6^3}$ of the overall memory. For a 3D example, usually the

block size is 10x10x10 which means the coeficients take only $\frac{1}{250}$ of overall memory.

4.2.3 Resilience against the memory error in quantization bin index array. In SZ, the quantization bin index array (to be called bin array for simplicity) is an array used to record how much the predicted value deviates from the original value for each data point. The element in the array is a positive integer if the data is predictable; otherwise, the element is 0, indicating that the data needs to be compressed/decompressed by unpredictable compression method. Obviously, if the bin array is corrupted by some memory error, the decompressed data will not be correct. So, the array is not resilient to memory error. Also, since the prediction is a critical stage that contributes the portion of the overall execution time, the likelihood of error happening during this stage is higher than other stages, thus we have to protect the bin array in this stage. Specifically, we carry out two different checksums on each block right after all the data inside the block are processed, such that we are able to detect and correct the possible corrupted data by double-checking the checksum values later (e.g., during the Huffman encoding stage).

5 ERROR TOLERANCE METHODOLOGY

Our resilient compression design is done in three aspects. First, we eliminate the data dependency between adjacent blocks; second, we use selective instruction duplication to ensure correct computation; third, we use checksums to detect and correct corrupted values caused by memory errors.

5.1 Blockwise independent design

In the following, we discuss how to eliminate the dependency between blocks, such that any silent error can be confined within a small block, improving the robustness significantly.

The key difference between the original SZ and our independent-block based compression is that we now treat each block of data as separately with each other. Specifically, we apply the prediction and quantization inside each block individually and make sure the compressed data of one block is totally independent with others'. This requires many changes to the original SZ development. For instance, we need to record the compressed size of each block after we finish the compression for that block. In addition, we also need to record the quantization bin array individually for each block and perform the Huffman encoding individually as per block. Both recording the bin array and Huffman encoding need to be done individually per block. The block-wise design in our solution is feasible for any structured mesh data with different dimensions (1D, 2D, and 3D), since block/segmentation design extracts/locates the data in each block based on their array indices in the space and performs the "isolated" data block by block.

Another significant advantage in the independent-block based compression design is that one can perform random-access decompression eficiently by specifying a specific region in space. To this end, we implement random-access support in our implementation, such that the decompression speed can be improved significantly if the user just wants to decompress a small region in the whole dataset. The corresponding experimental results will be presented in Section 6. Moreover, such an independent-block based compression also makes the parallelism of SZ much easier to port on many-core architectures, such as GPU. Also, if only a block of data is detected to be corrupted either because soft errors during data transferring or decompression, we just need to resend and decompress only that block.

5.2 Fault tolerant compression

Algorithm 2 Soft Error Resilient SZ Compression

Input: original input data (denote by □ □[]), user defined error bound (denoted by □).
Output: compressed data in byte and compressed □ □of blocked decompressed data

1: for each block, □□ [ ] with size □, of the input data do
2: Compute the regression coeficients
3: □ □□[□] ← □□[□] /*for silent error in input data*/
4: □ □ □[□] ← □□[□] /*for silent error in input data*/
5: end for
6: for each block (block □) of input data do
7: Sample and estimate □□ □ and □□ □
8: □ □ □ □ □[□] ← □□ □ ≤ □□ □? □ □ □
9: end for
10: for each block (block □) of input data do
11: memory_error_detect_correct(□ □[□], □ □□□ □ □□);
12: □() ← □ □ □ □ □[□] = □ □?1: □ □ □ □□□□();
13: for each data point, □ □ □in the data block do
14: □ □ □ ← □□ □() /*□ □(); instruction duplicated □()*/
15: Δ ← □ □□ − □ □ □;
16: q_bin ← |Δ|/□; /*Compute quantization bin based on error bound □*/
17: if (q_bin < bin_max) then
18: decmp ← □(pred', q_bin, □); /*Get decompressed data, □□() is instruction duplicated □()*/
19: if (| □ □□ − □ □ □ | > □□) then
20: Do unpredictable data compression; /*IEEE 754 binary compression*/
21: end if
22: □ □□[□]+=dcmp /*cksum for decompressed data of block □*/
23: else
24: Do unpredictable data compression; /*IEEE 754 binary compression*/
25: end if
26: □ □□[□] += □_□ □ /*for silent error in □_□ □[]*/
27: □ □ □[□] += □□□_□ □ /*for silent error in □_□ □[]*/
28: end for
29: end for
30: h_tree ← Huffman_tree(□_□ □[]) /*build Huffman tree*/
31: for each block of □_□ □[]do
32: memory_error_detect_correct(□_□ □[], □ □□_□□ □ □);
33: encoded_bin ← Huffman_encode(h_tree, q_bin[])
34: end for
35: compressed_bin ← Zstardard(encoded_bin) /*lossless compression*/
36: write_to_file(encoded_bin and unpredicatable data)
37: write_to_file(Zstandard(□ □□[])) /*Write checksum*/

We present our resilient compression method in Algorithm 2. We highlight the lines related to our fault tolerance design in blue font. Line 3 and 4 are calculating checksums for input data, in order to detect possible silent errors striking the input data later on. As we discussed in Section 4.2.1, we do not need to detect memory error in the input data during computations for regression coeficients and compression error estimation. We only detect whether the input data encounters memory errors before the data prediction gets started (line 11). If a data corruption is detected (by □ □□), it can be located and recovered by the pair of checksums (i.e., □ □□and □ □ □) applied on input data. Then, we protect the quantization bin array against memory errors (line 24 and 35). Line 29 and 40 are designed for detecting possible silent errors occurring in the decompression stage, to be detailed later. For the computation errors, instruction duplication can be used. Base on our analysis in Section 4.1, only data prediction (line 18) and calculating decompressed data (line 25) need to be protected by instruction duplication.

5.3 Fault tolerant decompression

The resilient SZ decompression is presented in Algorithm 3. Line 1-9 refers to the regular block-wise data decompression of SZ. Our resilience design starts from line 10. We constructed the checksums for each block and compressed the checksum array (i.e., □ □□□) by lossless compression (Zstd) during data compression. Accordingly, we need to decompress □ □□(line 10) before the error detection. Our idea is leveraging such checksums of decompressed data (i.e., □ □□□) constructed during compression to detect possible errors that happen during decompression. Specifically, after performing the data decompression for each block (line 1-9), our algorithm will calculate the corresponding checksums for each block of decompressed data and compare the checksums to □ □□□ (line 12-13). If they are not consistent, some errors must happen during the decompression. So, the algorithm will decompress this block by random-access decompression (line 14), meaning the compressed bytes are reloaded. If the checksum is consistent, we know some memory or computation error is detected (line 17). If inconsistent the second time, we know the silent error likely happens during pre-decompression procedures including lossless compression or data transferring which will be reported to users (line 19).

Algorithm 3 Soft Error Resilient SZ Decompression

Input: The SZ compressed file in byte (cmp_data) and compressed □ □for blocked decompressed data □ □_□ □□□.
Output: Decompressed data with bounded error compared to original data.

```
1:  zstd_decmp ← Zstandard_decompress(cmp_data)
2:  for each block do
3:      □_□ □[]□ ← Huffman_decoding(zstd_decmp)
4:      □ □_□ □ □[]□ ← was predicted by Lorenzo ? lor_dec() : reg_dec()
5:  end for
6:  □ □□□[] ← Zstandard_decompress(□ □_□ □□□) 7:
for each block of decompressed data (block □) do 8:
        □ □□□ ← □ □_□ □ □[]□ for block □
9:      if □ □□□ ≠ □ □□□[□] then
10:         Reexecute line 3-4 for block □ /*random-access decompression*/
11:         □ □□□ ← □ □_□ □ □[]□ for block □
12:         if □ □□□ = □ □□□[□] then
13:             Log: memory/computation error detected but corrected
14:         else
15:             Log: silent error in pre-decompression procedure.
                consider resend block □; Return
16:         end if
17:     end if
18: end for
```

5.4 Avoiding round off errors in checksums

Since the input data and the decompressed data are both floating point numbers, round off errors in the checksums may introduce inaccurate memory error corrections. To avoid the impact of round off error, we treat the floating point numbers as unsigned 32-bit integers and then calculate checksums based on these integers. We first describe how the checksum is performed on the 32-bit single-precision floating point data as an example and then discuss how to extend it to 64-bit double-precision floating point values.

Given a data block of 32-bit floating point values, for each element, we put all its 32 bits in a temporary variable and treat the bits in that variable as a 64-bit unsigned integer with the first 32 bits being flushed to 0. We then add that integer to the checksum which is also a 64-bit unsigned integer. Finally, we get the checksum represented by a 64-bit unsigned integer for this data block. Notice that the "checksum" here is not equal or approximates to the real sum of the data block because it is calculated based on integer interpretation of the bits instead of floating point. Thus, it is immune to NaN/Inf issues that happens only to floating point numbers. Using the 64-bit unsigned integer representation, we can have the checksum hold up to $(2^{32}+1)$ 32-bit unsigned integers without overflow because the maximum 64-bit unsigned integer $(2^{64}-1)$ divided by maximum 32-bit unsigned integer $(2^{32}-1)$ is equal to $(2^{32}+1)$. That is fairly enough to totally avoid the overflow since each data block in SZ has only 1000 data points (such as 10×10×10 block) in general. With all these techniques, we can provide bit-level error detection and correction. The main difference from Demmel's work [20] is that we are actually doing integer-based summation instead of the sum based on floating point numbers.

To extend to 64-bit double precision numbers, we just need to treat each double value as two 32-bit unsigned integers. So it is reduced to the above case.

5.5 Impact to compression ratio without protecting regression and sampling

As mentioned previously, we do not protect the computation in regression and sampling in that the errors during this period would not affect the correctness of decompressed data and just have tiny impact to the compression ratios. In what follows, we derive theoretically the upper bound of the compression ratio decrease affected by the computation errors happening during the regression or sampling. We denote the compression ratio of SZ in error free run by $□_0$; the number of data blocks by □. For simplicity, we assume that the compression ratio for each block is identical with each other. In the worst case, the error in regression or sampling will at most reduce the compression ratio to be 1, which means that it does not reduce the size of that block of data. Consequently, we can derive the maximum compression ratio decrease as CR_decrease $= (\frac{□_0-1}{□_0+□-1})$×100%. Based on the above equation, the upper bound of compression ratio decrease depends on the error free compression ratio and the block size. For example, if the block size is set to 6×6×6 and the compression ratio is 10, and if the input data is around 864 MB, then there will be 10^6 data blocks. The compression ratio decrease would be bounded within $\frac{10-1}{10-1+10^6} < 0.1\%$, which is negligible to the overall compression ratios.

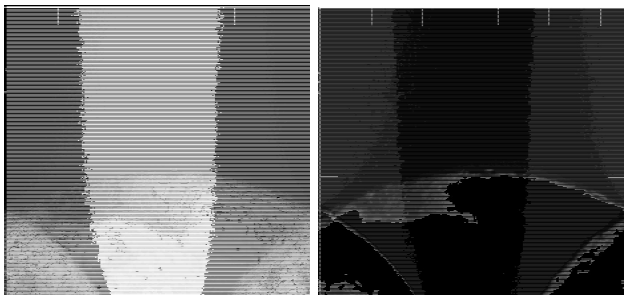
6 EXPERIMENTAL EVALUATION

6.1 Experimental setup

In this subsection, we describe how we set the experiments in our evaluation, including applications, error injections, and experimental environment.

6.1.1 Applications. We evaluate our resilient error-bounded SZ compressor on three real scientific datasets: NYX, Hurricane, and SCALE-LETKF (SL for short). We also evaluate our fault tolerant compressor using 20 Pluto images provided by Plantary Data System (PDS) [6]. Those images were taken by New Horizons space probe [4] in aerospace which is an error-prone environment because of potential impact of cosmic rays. The description to these datasets is presented in Table 1. Each application involves multiple one or multiple fields, such as dark matter density (density of dark matter), baryon density (density of baryon), and temperature in

the simulated cosmology space, and velocities in different dimensions (denoted as velocity-X,velocity-Y,velocity-Z for NYX [44] or denoted as U,V,W for climate/weather [28, 51]). For the Pluto image data, we perform the error-bounded lossy compression such that the visual quality can be maintained very well, as illustrated in Figure 2. To indicate the reconstructed data has a high visual quality, we calculate both peak signal-to-noise ratio (PSNR) and structural similarity metric (SSIM) [50] between original data and decompressed data. PSNR and SSIM are two commonly used assessment metrics for visual quality. Our calculation shows that the PSNR and SSIM are both very high for the reconstructed data when error bound is 1E-4, indicating a fairly high precision in visual quality in this test-case.



(a) Original image (b) SZ decompressed image

Figure 2: Visualization of Original Data vs. Decompressed Data (Pluto photo taken by New Horizons [4]; SZ compression using Value-range based error bound of 1E-4): PSNR = 84.8 dB, SSIM = 0.9936, NRMSE = 5.766E-5

6.1.2 Error injections with two modes.

Evaluation mode A - source-code level error injection. Like most ABFT work [17, 34], we inject errors at the source code level and only inject errors to the main data structures. Specifically, as for the memory errors in input data and quantization bin array, we randomly choose an index from the array and randomly flip a bit of the selected data value during the compression. Thus, we simulate memory error randomness both in time and location. We inject them after the checksums are applied on input data (i.e., □ □□□ and □ □ □□). To simulate the computation errors when calculating regression coeficients, sampling and estimating compression error of Lorenzo and regression, we randomly select a data point in a random block and then change its value by injecting a random bitflip error. We exclude the evaluation of computation errors in prediction as it is already protected by instruction redundancy.

Evaluation mode B - system level error injection. Besides the evaluation mode A (memory errors happens only to the data we protected), we also follow a Checkpoint-based Fault Injection (CFI) [10] model to inject random error(s) to the whole memory consumed during the compression. We adopt a system-level checkpointing toolkit - Berkeley Lab Checkpoint/Restart (BLCR) [3], which can dump the whole memory of a running process to disk as a checkpoint and then restart its execution from that checkpoint. In our experiment, we select a random time stamp during the whole compression period. Then, we set a checkpoint by saving the whole memory at that time stamp using BLCR and kill the process. We then inject a random bitflip error in the checkpoint file and restart by the bit-flipped checkpoint. We inject 1, 2 or 3 errors and perform 500 runs per test for both fault tolerant SZ and the original SZ.

6.1.3 Experimental Environment.

We run experiments on a supercomputer (Bebop) [1]. Inside each computing node are two Intel Xeon E5-2695 v4 processors totalling 36 cores. The POSIX I/O [52] with mode, file-per-process, is used for parallel data reading and writing. We implement our solution in SZ's source code and call it ftrsz (or FT-SZ) in the following text. We alter the order of value additions in the duplicated computation of data prediction, which can effectively prevent the compiler from overlooking this operation, and the execution time overhead can thus be measured correctly.

6.2 Comparison with state-of-the-art

We compare our error resilient SZ with the classic recomputation method since there is no other resilience work on lossy compressor to the best of our knowledge. We did not compare with instruction redundancy because it cannot protect against memory errors. In the recomputation method, we run the SZ compression or decompression twice and check if the two compression results are identical. We use SZ in the recomputation method as it exhibits both competitive compression ratio and speed from among all error-bounded lossy compressors (as verified in exhisting literature [21, 31, 35, 45, 48]). In spite of the great generality of the recomputation based resilience, it brings very high overhead in terms of execution time, which will be presented in Section 6.4.2. Specifically, the SZ-based recomputation method causes significantly higher cost (100% versus 5%□20% as shown in Figure 5 and Figure 6) and twice amount of original compute resources, so it is the least users want to do for their simulations in practice. As such, we focus on the comparison of our fault-tolerant SZ versus the original SZ in the following text.

6.3 Evaluation of Independent-block Compression

We first evaluate our designed independent-block based SZ compression (a.k.a., random-access based compression).

6.3.1 Exploration of The Best Block Size.

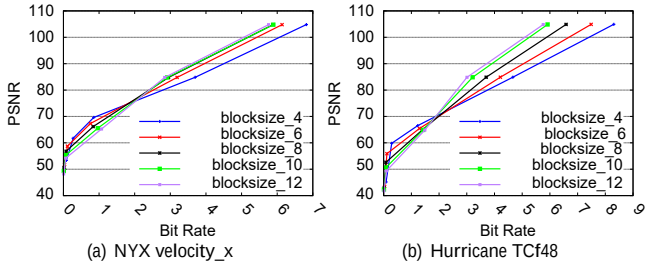
It is important to determine an appropriate block size in our independent-block based compression framework. We determine the best block size by a comprehensive analysis in terms of rate-distortion with masses of experiments using different block sizes, as the optimal block size is hard to find for different datasets by theory.

We evaluate the compression results using the block size of 4x4x4 through 20x20x20. We exemplify the rate-distortion with cosmological NYX simulation data (velocity_x field) and climate hurricane simulation data (TCf48 field) with five different block sizes in Figure 3. As shown in the figure, small block sizes (such as 4x4x4 and 6x6x6) may lead to high PSNR in the cases with low bit-rates (such as ≤2); large block sizes (such as 8x8x8 □ 12x12x12) would be clearly better than the small block sizes on high bit-rates. The reason is explained as follows. For the over-small block sizes such as 4x4x4, the overhead of storing the regression-coeficients appears relatively high compared to the overall compressed size. For the over-large block sizes such as 20x20x20, the linear-regression based predictor cannot get a good fitting for the data. Based on

Table 1: Description to scientific datasets used in our evaluation

Dataset	# Fields	Dimensions	Precision	Science	Example Fields
NYX [44]	6	512X512X512	float	Cosmology	Dark matter density, Baryon density, temperature, vel-X, vel-Y, vel-Z
Hurricane Isabel[28]	13	100X500X500	float	Climate	QCLOUD, QGRAUP, QICE, QSNOW, PRECIP, U, V, W, etc.
SCALE-LETKF (SL) [51]	6	98X1200X1200	float	Weather	W, V, U, QR, QC, PRES
NASA: Pluto [6]	1	1028X1024	float	Aerospace	all are 2D image files with different angles or distances.

our experiments with multiple simulation data, we set the block size to 10x10x10 in our implementation because it has much better compression ratios (i.e., low bit-rate) in the hard-to-compress cases than other block sizes, while it exhibits comparative compression ratios with other block sizes in the cases with low bit-rates.



(a) NYX velocity_x (b) Hurricane TCf48

Figure 3: Rate distortion with different block sizes

6.3.2 Evaluating independent-block decompression. The biggest advantage of the independent-block based implementation is very fast decompression speed if the users just want to extract a small sub-block of data. Moreover, as we discussed in Section 5.3, this design can also help correct the errors very quickly upon a detection of problematic blocks by checksums. In Figure 4, we present the decompression times with different data sizes compared to the whole dataset. The x-axis indicates the ratio of the decompressed data size to the whole data size. In the figure, we observe that the decompression time decreases approximately linearly with decreasing data size in the decompression, which confirms the high eficiency of random-access decompression.

6.4 Error free experimental results

One key indicator is how much overhead (including compression ratio overhead and execution time overhead) would be introduced by the silent error detection in the compressor.

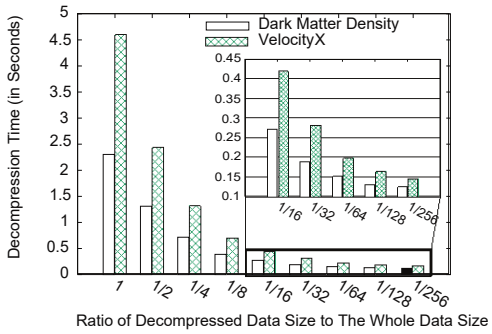


Figure 4: Eficiency of random access decompression

Table 2: Compression ratio degradation of random-access SZ (rsz) and fault-tolerant random-access SZ (ftrsz)

error bound:	1E-3	1E-4	1E-5	1E-6	1E-3	1E-4	1E-5	1E-6
	NYX				Hurricane			
sz:	17.0	7.7	4.6	3.1	8.4	5.1	3.1	2.4
rsz decrease:	8.7%	3.7%	3.1%	3.2%	8.5%	4.7%	1.2%	1.5%
ftrsz decrease:	10.7%	4.7%	3.7%	3.6%	9.3%	5.2%	1.6%	1.7%
	SCALE-LETKF (SL)				Pluto			
sz:	19.1	8.7	5.2	3.7	7.1	4.0	3.4	3.2
rsz decrease:	23.6%	21.3%	13.5%	9.1%	4.2%	0.3%	0.1%	0%
ftrsz decrease:	24.9%	21.9%	13.9%	9.4%	5.6%	0.8%	0.1%	0%

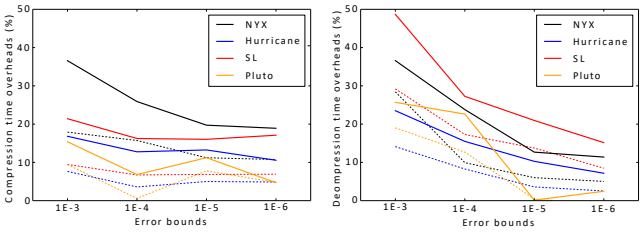
6.4.1 Compression ratio overhead. We evaluate the the compression ratio overhead in fault-tolerant compression mehtods. Table 2 presents the compression ratios of the original SZ (denoted as sz) and the relative decreases of compression ratios under the independent-block based SZ (or random-based SZ, abbreviated as rsz) and fault-tolerant random-access SZ (denoted as ftrsz), respectively. It is observed that our proposed solution incurs only 0–10.7% degradation on compression ratio for NYX, Hurricane and Pluto data, and the degradation level decreases with decreasing error bounds. The SL dataset exhibits 9.4–24.9% compression ratio degradation, which mainly comes from the overhead introduced by the random-access design. The general reason for the degradation of compression ratios that we store the checksum □ □□ during the compression in order to verify the correctness of the decompressed data. In principle, the key reason for SL dataset getting more compression ratio degradation is that the original SZ uses data across blocks in its prediction while the random-access SZ can only uses the data confined within each block in the prediction, which may cause lower prediction accuracy especially when the simulation data is relatively smooth in space.

6.4.2 Execution time overhead. We evaluate the time overheads introduced by our fault tolerance codes added to SZ when there are no errors. We show the results in both compression and decompression in Figure 5. We can see from Figure 5 that in most cases, the rsz and ftrsz incur about 5–20% overheads in compression time and 2–30% overheads in decompression time. Such time overhead, actually, are negligible compared to the total I/O time on a PFS because of potential I/O bottleneck, which will be demonstrated in the end of this section.

We also compare our error resilience solution with recomputation-based resilience. The compression and decompression time overheads of recomputation-based resilience is shown in Figure 6. As we can see, the experimental overheads are quite consistent with the analytical complexity overheads which is around 100%. Notice that recomputation-based resilience only provides error detection ability. If error correction is needed, a third execution will be incurred which leads to roughly 200% overheads.

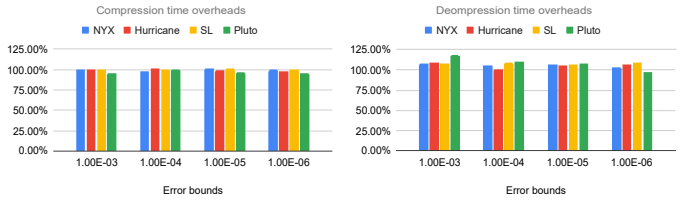
Table 3: Percentage of runs whose maximum absolute error is within error bounds in sz and ftrsz

	injecting errors in input data				injecting errors in quantization bin array							
	Successful runs with correct decompressed data				Successful runs with correct decompressed data				Normal runs without core-dump segmentation faults			
error bounds:	1E-3	1E-4	1E-5	1E-6	1E-3	1E-4	1E-5	1E-6	1E-3	1E-4	1E-5	1E-6
sz	60%	57%	49%	48%	3%	1%	1%	0%	34%	34%	49%	54%
ftrsz	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%



(a) Compression (b) Decompression

Figure 5: Compression time and decompression time overheads. Dash lines are random access SZ; solid lines are fault tolerant random access SZ.



(a) Compression (b) Decompression

Figure 6: Compression time and decompression time overheads of re-computation based resilience.

6.5 Error injected experimental results

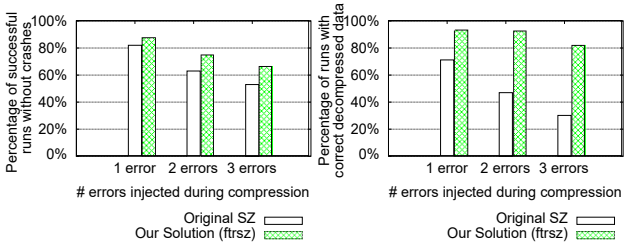
6.5.1 Resilience against memory errors in input and quantization bin array (evaluation mode A). We first inject memory errors into the input array and bin array to verify that our proposed solution can still ensure the decompressed data within user's error bounds.

In this experiment, we observe that various fields exhibit similar results. As such, we present the results based on the field of dark matter density in NYX dataset as an example. For every error bound, we repeat running sz and ftrsz for 100 times, each with randomly injected memory errors in input and quantization bin array.

As shown in Table 3, our proposed fault tolerance solution can always yield correct decompressed results when the memory errors are injected in input data or quantization bin array. The 100% correctness of the decompressed data under ftrsz also means that our solution is immune to the round-off errors. In comparison, for the original SZ, we can see that only 48□60% runs can yield error bounded decompressed data when the input data experiences memory errors. As the memory error corrupts a value in the bin array, the situation gets worse because some of the memory errors may cause core-dump segmentation fault, which happens in the case that the corrupted values turn out to be a fresh value such that it is beyond the range of the constructed Huffman tree. As shown in the right side of Table 3, under the original SZ compression, only 34□54% runs can complete without segmentation faults; and only 0-3% runs can complete with correct decompressed data.

As for the extra time overheads introduced by the detection/correction of errors in our fault tolerance method, we conduct error injected experiments for all three datasets. The extra overheads compared to ftrsz in an error-free case are all less than 1% for any error bound. This is because the case with injected errors only incurs one more block of checksum calculation, which is negligible to the overall execution time.

6.5.2 Resilience against memory errors happening anywhere (evaluation mode B). Figure 7 presents the experimental results of our solution (ftrsz) against the original SZ in the evaluation mode B (i.e., by injecting the errors into the whole memory during the compression). It is observed that our solution can improve the percentage of successful non-crash runs by 10%□20%, and improve the percentage of the runs with correct decompression results by 30%□170%. Our solution can substantially reduce the crash runs because we protect the bin arrays, which may run into core-dump segmentation faults when being injected errors, as shown in Table 3. In addition, as shown in Figure 7 (b), when injecting one and two memory errors respectively, about 92% of running cases lead to correct decompressed data (with guaranteed error bound) under our solution, while the original SZ suffers very low percentage (71.2% and 47%, respectively). For our solution, the 8% failed cases with incorrect decompression data are likely due to the error injec-tion before the checksum execution at the beginning period, which means the checksum is calculated based on corrupted input data. Thus, it will not be able to detect future memory errors.



(a) Runs without crashes (b) silent error

Figure 7: Experimental results using evaluation mode B

6.5.3 Resilience against computation errors during compression. As discussed in Section 4.1.1, the computations of regression coeficients, sampling and estimating compression error are error resilient though computation errors will impact the compression ratio. Figure 8 shows our experimental results about the impact to compression ratios. Computation errors are randomly injected and each experiment is repeated 50 times. The compression ratio decrease is calculated by taking the lowest compression ratio among 50 trials. As can been seen, the compression ratio decrease is within

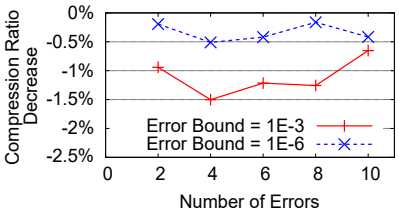


Figure 8: Compression ratio decrease with cmput. errors

2% for up to 10 computation errors injected under the error bound of 1E-6 or 1E-3. The compression ratios in an error-free case are 4.8023 and 1.8112 for these two error bounds, respectively.

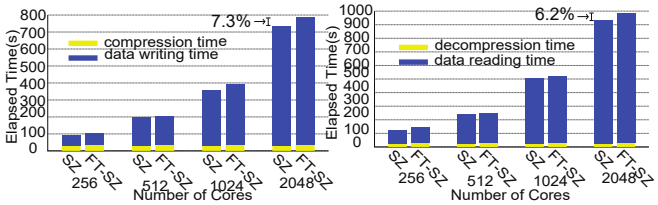
For each run of decompression, we injected one computation error to a random block and noted all the errors can be 100% detected by checksum and corrected by re-executing decompression for that block. Again, the extra overheads compared to fault tolerant random access SZ in error-free cases are all less than 1% for all datasets in all error bounds.

6.6 Parallel experimental results

As mentioned in Section 1, error-bounded lossy compressor can be particularly helpful for WAN data transfers (e.g., through Globus [23]), in which large amounts of data are loaded from disk (or parallel file system (PFS)), compressed at the data sender in parallel, and then decompressed and written to the disk or PFS by the receiver in parallel. Compression can accelerate the overall data transferring throughput significantly because WAN bandwidth is in general unstable and relatively low. According to [40], Globus bandwidth on WAN is only several GB/s even with a high concurrency (transferring 500 files concurrently), which is still far less than the parallel compression throughput that can reach tens to hundreds of GB/s depending on the number of cores.

In the following text, we evaluate the overhead of our resilient compressor over the total data reading/writing time (including compression and decompression time) by running the parallel compression/decompression on a supercomputer, and then evaluate the performance gain in data transferring by a simulation.

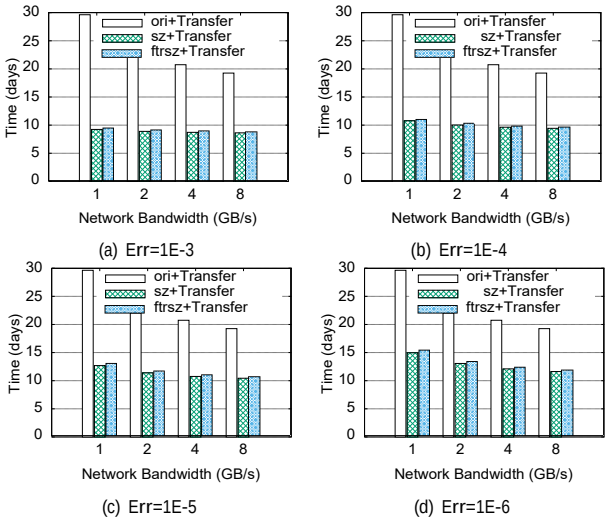
We evaluate the I/O performance with breakdown of the execution times (compression/decompression time + data writing/reading time) by processing the NYX dataset in parallel on the supercomputer Bebop [1] with an error bound of 1E-4. We run a weak-scaling experiment with different execution scales (256–2,048 cores), in which each rank kept the same data size (3GB) to process. Results are shown in Figure 9. The performance results are fairly stable with low variance since the total time in each test is relatively long (100–800 seconds) , and all the performance-related factors including compression ratio, compression time, and disk I/O bandwidth are deterministic or stable. As for the total data dumping time, it is observed that our error-resilient SZ incurs only 7.3% overhead at the scale of 2,048 cores. Our error-resilient SZ has only 6.2% overhead on the data dumping performance when using 2k cores to read and decompress data. The key reason for the very limited overall overhead is that the total I/O performance is dominated by compression ratio because of the I/O bottleneck of the PFS.



(a) Data Dumping (b) Data Loading

Figure 9: Performance of data dumping/loading (sz vs. ftrsz): yellow parts represent the compression and decompression time, and blue part represents disk I/O time

From Globus network bandwidth characterization results between different end-points, we can also get simulated transferring times with and without our fault-tolerant lossy compression. The data transferring simulation involves all steps from loading data from sender's disk to writing data in the receiver's disk. The simulation is based on the real performance and profiled by running SZ to compress NYX dataset on the Bebop supercomputer (e.g., disk reading speed is 1.5GB/s, disk writing speed is 1.2GB/s, 2048-core parallel compression throughput is about 150GB/s, and compression ratios are shown in Table 2). The only emulated resource is the network bandwidth: 1GB/s to 8GB/s, which is also based on real-world experiments [40]. Figure 10 clearly shows that when transferring 1PB of data on Globus across different sites, the compression techniques can significantly lower the overall transferring time by 50%–68% (from 30 days to 10–15 days), depending on the status of network bandwidth. Ftrsz introduces totally negligible overhead (less than 2%) compared with the original SZ compressor. In fact, many of today's supercomputers such as Theta [2] and Summit [5] have I/O bandwidth higher than 100+GB/s. If the aggregated disk reading and writing bandwidths are both 50GB/s, our simulation shows that the overall data transferring time under ftrsz will be reduced to 1 day from 12 days. Note that ftrsz also protects the data against different types of silent errors.



(a) Err=1E-3 (b) Err=1E-4

(c) Err=1E-5 (d) Err=1E-6

Figure 10: 1PB Data Transferring Time on Globus

7 CONCLUSION

In this paper, we propose a novel resilient strategy for the SZ lossy compressor. We develop an independent-block based compression model for SZ to improve its robustness. We analyze each subroutine of the SZ framework and then design a series of fault tolerance strategies for the fragile code segments. We perform the evaluation by processing three well-known scientific datasets on a cluster with up to 2048 cores and observe the following insights:

- Our solution can control the time overhead to about 10%, with a degradation of compression ratio limited within 5%.
- When injecting one and two silent errors respectively during the compression, our solution can have about 92% running cases get correct decompressed data (with guaranteed error bound), which is significantly higher than that of the original SZ (71.2% & 47%, respectively).
- Our solution suffers from very low fault-tolerance overhead. The overall time increases only 7.3% when loading + compressing the data on a supercomputer's PFS, and increases only 6.2% when decompressing + writing the data.
- Our solution introduces ≤2% overhead when transferring 1PB of data on WAN/Globus, and can reduce the transferring time by 50%–68% compared with transferring original data.

In the future work, we plan to explore resilient strategies for more error-bounded lossy compression models such as block-transform based model [38] and Higher-order singular value decomposition (HOSVD) based model [13].

8 ACKNOWLEDGMENTS

This research was supported by the Exascale Computing Project (ECP), Project Number: 17-SC-20-SC, a collaborative effort of two DOE organizations – the Office of Science and the National Nuclear Security Administration, responsible for the planning and preparation of a capable exascale ecosystem, including software, applications, hardware, advanced system engineering and early testbed platforms, to support the nation's exascale computing imperative. The material was supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, under contract DE-AC02-06CH11357, and supported by the National Science Foundation under Grant No. 1617488, Grant No. 1619253 and Grant No. 2003709. We acknowledge the computing resources provided on Bebop, which is operated by the Laboratory Computing Resource Center at Argonne National Laboratory.

REFERENCES

[1] [n.d.]. ANL Bebop. Retrieved January 23, 2020 from https://www.lcrc.anl.gov/systems/resources/bebop/
[2] [n.d.]. ANL Theta supercomputer. https://www.alcf.anl.gov/support-center/theta
[3] [n.d.]. Berkeley Lab Checkpoint/Restart (BLCR) for LINUX. Retrieved January 23, 2020 from https://crd.lbl.gov/departments/computer-science/CLaSS/research/BLCR
[4] [n.d.]. New Horizons: The First Mission to the Pluto System and the Kuiper Belt. Retrieved January 23, 2020 from nasa.gov/newhorizons
[5] [n.d.]. ORNL Summit supercomputer. https://www.olcf.ornl.gov/summit/
[6] [n.d.]. PDS: The Planetary Data System. Retrieved January 23, 2020 from https://pds.jpl.nasa.gov
[7] [n.d.]. Zstandard. Retrieved January 23, 2020 from https://github.com/facebook/zstd/releases
[8] Mark Ainsworth, Ozan Tugluk, Ben Whitney, and Scott Klasky. 2018. Multilevel Techniques for Compression and Reduction of Scientific Data—the Univariate Case. 19, 5-6 (2018), 65–76.
[9] A Alekseev, A Kiryanov, A Klimentov, T Korchuganova, V Mitsyn, D Oleynik, A Smirnov, S Smirnov, and A Zarochentsev. 2020. Scientific Data Lake for High Luminosity LHC project and other data-intensive particle and astro-particle physics experiments. Journal of Physics: Conference Series 1690 (dec 2020), 012166. https://doi.org/10.1088/1742-6596/1690/1/012166
[10] Cyrille Artho, Kuniyasu Suzaki, Masami Hagiya, Watcharin Leungwattanakit, Richard Potter, Eric Platon, Yoshinori Tanabe, Franz Weitl, and Mitsuharu Yamamoto. 2015. Using checkpointing and virtualization for fault injection. International Journal of Networking and Computing 5, 2 (2015), 347–372.
[11] Allison H. Baker, Haiying Xu, John M. Dennis, Michael N. Levy, Doug Nychka, Sheri A. Mickelson, Jim Edwards, Mariana Vertenstein, and Al Wegener. 2014. A methodology for evaluating the impact of data compression on climate simulation data. In The 23rd International Symposium on High-Performance Parallel and Distributed Computing, HPDC'14, Vancouver, BC, Canada - June 23 - 27, 2014, Beth Plale, Matei Ripeanu, Franck Cappello, and Dongyan Xu (Eds.). ACM, 203–214. https://doi.org/10.1145/2600212.2600217
[12] Allison H Baker, Haiying Xu, John M Dennis, Michael N Levy, Doug Nychka, Sheri A Mickelson, Jim Edwards, Mariana Vertenstein, and Al Wegener. 2014. A methodology for evaluating the impact of data compression on climate simulation data. In Proceedings of the 23rd international symposium on High-performance parallel and distributed computing. 203–214.
[13] R. Ballester-Ripoll, P. Lindstrom, and R. Pajarola. 2020. TTHRESH: Tensor Compression for Multidimensional Visual Data. IEEE Transactions on Visualization and Computer Graphics 26, 9 (2020), 2891–2903. https://doi.org/10.1109/TVCG.2019.2904063
[14] Mikaël Capelle, Marie-José Huguet, Nicolas Jozefowiez, and Xavier Olive. 2019. Optimizing ground station networks for free space optical communications: maximizing the data transfer. Networks 73, 2 (2019), 234–253.
[15] Franck Cappello, Sheng Di, and Ali Murat Gok. 2020. Fulfilling the Promises of Lossy Compression for Scientific Applications. In Driving Scientific and Engineering Discoveries Through the Convergence of HPC, Big Data and AI, Jeffrey Nichols, Becky Verastegui, Arthur 'Barney' Maccabe, Oscar Hernandez, Suzanne Parete-Koon, and Theresa Ahearn (Eds.). Springer International Publishing, Cham, 99–116.
[16] 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. https://doi.org/10.1177/1094342019853336 arXiv:https://doi.org/10.1177/1094342019853336
[17] Jieyang Chen, Xin Liang, and Zizhong Chen. 2016. Online algorithm-based fault tolerance for cholesky decomposition on heterogeneous systems with gpus. In 2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS). IEEE, 993–1002.
[18] Zizhong Chen. 2008. Extending algorithm-based fault tolerance to tolerate fail-stop failures in high performance distributed environments. In 2008 IEEE International Symposium on Parallel and Distributed Processing. IEEE, 1–8.
[19] Zhengzhang Chen, Seung Woo Son, William Hendrix, Ankit Agrawal, Wei-keng Liao, and Alok Choudhary. 2014. NUMARCK: machine learning algorithm for resiliency and checkpointing. In SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. IEEE, 733–744.
[20] James Demmel and Hong Diep Nguyen. 2013. Fast reproducible floating-point summation. In 2013 IEEE 21st Symposium on Computer Arithmetic. IEEE, 163–172.
[21] Sheng Di and Franck Cappello. 2016. Fast error-bounded lossy HPC data compression with SZ. In 2016 ieee international parallel and distributed processing symposium (ipdps). IEEE, 730–739.
[22] Thomas E Fornek. 2017. Advanced photon source upgrade project preliminary design report. Technical Report. Argonne National Laboratory (ANL)(United States). Funding organisation ….
[23] Ian Foster and Carl Kesselman. 1997. Globus: A metacomputing infrastructure toolkit. The International Journal of Supercomputer Applications and High Performance Computing 11, 2 (1997), 115–128.
[24] Ali Murat Gok et al. 2018. PaSTRI: A novel data compression algorithm for two-electron integrals in quantum chemistry. In IEEE International Conference on Cluster Computing (CLUSTER). 1–11.
[25] Salman Habib, Vitali Morozov, Nicholas Frontiere, Hal Finkel, Adrian Pope, and Katrin Heitmann. 2013. HACC: extreme scaling and performance across diverse architectures. In SC'13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis. IEEE, 1–10.
[26] Peter H Hochschild, Paul Turner, Jeffrey C Mogul, Rama Govindaraju, Parthasarathy Ranganathan, David E Culler, and Amin Vahdat. 2021. Cores that don't count. In Proceedings of the Workshop on Hot Topics in Operating Systems. 9–16.
[27] Kuang-Hua Huang and Jacob A Abraham. 1984. Algorithm-based fault tolerance for matrix operations. IEEE transactions on computers 100, 6 (1984), 518–528.
[28] Hurricane ISABEL simulation data. 2016. https://www.earthsystemgrid.org/dataset/isabeldata.html. Online.

[29] Lawrence Ibarria, Peter Lindstrom, Jarek Rossignac, and Andrzej Szymczak. 2003. Out-of-core compression and decompression of large n-dimensional scalar fields. In Computer Graphics Forum, Vol. 22. Wiley Online Library, 343–348.

[30] Adam M Jacobs. 2013. Reconfigurable fault tolerance for space systems. University of Florida.

[31] Sian Jin, Jesus Pulido, Pascal Grosset, Jiannan Tian, Dingwen Tao, and James Ahrens. 2021. Adaptive Configuration of In Situ Lossy Compression for Cosmology Simulations via Fine-Grained Rate-Quality Modeling. In Proceedings of the 30th ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC2021).

[32] Rajkumar Kettimuthu, Zhengchun Liu, David Wheeler, Ian Foster, Katrin Heitmann, and Franck Cappello. 2018. Transferring a petabyte in a day. Future Generation Computer Systems 88 (2018), 191–198.

[33] Sriram Lakshminarasimhan, Neil Shah, Stephane Ethier, Scott Klasky, Rob Latham, Rob Ross, and Nagiza F Samatova. 2011. Compressing the incompressible with ISABELA: In-situ reduction of spatio-temporal data. In European Conference on Parallel Processing. Springer, 366–379.

[34] Xin Liang, Jieyang Chen, Dingwen Tao, Sihuan Li, Panruo Wu, Hongbo Li, Kaiming Ouyang, Yuanlai Liu, Fengguang Song, and Zizhong Chen. 2017. Correcting soft errors online in fast fourier transform. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. 1–12.

[35] Xin Liang, Sheng Di, Dingwen Tao, Sihuan Li, Shaomeng Li, Hanqi Guo, Zizhong Chen, and Franck Cappello. 2018. Error-controlled lossy compression optimized for high compression ratios of scientific datasets. In 2018 IEEE International Conference on Big Data (Big Data). IEEE, 438–447.

[36] Xin Liang, Hanqi Guo, Sheng Di, Franck Cappello, Mukund Raj, Chunhui Liu, Kenji Ono, Zizhong Chen, and Tom Peterka. 2020. Toward Feature-Preserving 2D and 3D Vector Field Compression. In 2020 IEEE Pacific Visualization Symposium (PacificVis). 81–90. https://doi.org/10.1109/PacificVis48177.2020.6431

[37] SLim. 2009. A fault tolerant parallel computing architecture for remote sensing satellites. (2009).

[38] Peter Lindstrom. 2014. Fixed-rate compressed floating-point arrays. IEEE transactions on visualization and computer graphics 20, 12 (2014), 2674–2683.

[39] Peter Lindstrom and Martin Isenburg. 2006. Fast and eficient compression of floating-point data. IEEE transactions on visualization and computer graphics 12, 5 (2006), 1245–1250.

[40] Y. Liu, Z. Liu, R. Kettimuthu, N. Rao, Z. Chen, and I. Foster. 2019. Data Transfer between Scientific Facilities – Bottleneck Analysis, Insights and Optimizations. In 2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID). 122–131. https://doi.org/10.1109/CCGRID.2019.00023

[41] Tao Lu, Qing Liu, Xubin He, Huizhang Luo, Eric Suchyta, Jong Choi, Norbert Podhorszki, Scott Klasky, Mathew Wolf, Tong Liu, et al. 2018. Understanding and modeling lossy compression schemes on HPC scientific data. In 2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS). IEEE, 348–357.

[42] Gabriel Marcus, Zhirong Huang, Yuantao Ding, Tor Raubenheimer, Lanfa Wang, Marco Venturini, Paul Emma, and Ji Qiang. 2015. High fidelity start-to-end numerical particle simulations and performance studies for LCLS-II. In 37th International Free Electron Laser Conference. TUP007.

[43] Nor Rizuan Mat Noor and Tanya Vladimirova. 2013. Parallelised fault-tolerant Integer KLT implementation for lossless hyperspectral image compression on board satellites. In 2013 NASA/ESA Conference on Adaptive Hardware and Systems (AHS-2013). IEEE, 115–122.

[44] Nyx. 2013. https://ccse.lbl.gov/Research/NYX/. Online.

[45] Andrew Poppick, Joseph Nardi, Noah Feldman, Allison H. Baker, Alexander Pinard, and Dorit M. Hammerling. 2020. A statistical analysis of lossily compressed climate model data. Computers & Geosciences 145 (2020), 104599. https://doi.org/10.1016/j.cageo.2020.104599

[46] Abhishek Rhisheekesan, Reiley Jeyapaul, and Aviral Shrivastava. 2019. Control Flow Checking or Not? (For Soft Errors). ACM Trans. Embed. Comput. Syst. 18, 1, Article 11 (Feb. 2019), 25 pages. https://doi.org/10.1145/3301311

[47] Naoto Sasaki, Kento Sato, Toshio Endo, and Satoshi Matsuoka. 2015. Exploration of lossy compression for application-level checkpoint/restart. In 2015 IEEE International Parallel and Distributed Processing Symposium. IEEE, 914–922.

[48] Dingwen Tao, Sheng Di, Zizhong Chen, and Franck Cappello. 2017. Significantly improving lossy compression for scientific data sets based on multidimensional prediction and error-controlled quantization. In 2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS). IEEE, 1129–1139.

[49] Dingwen Tao, Shuaiwen Leon Song, Sriram Krishnamoorthy, Panruo Wu, Xin Liang, Eddy Z Zhang, Darren Kerbyson, and Zizhong Chen. 2016. New-sum: A novel online abft scheme for general iterative methods. In Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing. 43–55.

[50] Zhou Wang, Alan C Bovik, Hamid R Sheikh, and Eero P Simoncelli. 2004. Image quality assessment: from error visibility to structural similarity. IEEE transactions on image processing 13, 4 (2004), 600–612.

[51] SCALE-LETKF weather model. [n.d.]. https://github.com/gylien/scale-letkf. Online.

[52] Brent Welch. 2005. Posix io extensions for hpc. In Proceedings of the 4th USENIX Conference on File and Storage Technologies (FAST).

[53] Panruo Wu, Nathan DeBardeleben, Qiang Guan, Sean Blanchard, Jieyang Chen, Dingwen Tao, Xin Liang, Kaiming Ouyang, and Zizhong Chen. 2017. Silent data corruption resilient two-sided matrix factorizations. In Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. 415–427.

[54] Panruo Wu, Chong Ding, Longxiang Chen, Teresa Davies, Christer Karlsson, and Zizhong Chen. 2013. On-line soft error correction in matrix–matrix multiplication. Journal of Computational Science 4, 6 (2013), 465–472.

[55] X. Zou, T. Lu, W. Xia, X. Wang, W. Zhang, S. Di, D. Tao, and F. Cappello. 2019. Accelerating Relative-error Bounded Lossy Compression for HPC datasets with Precomputation-Based Mechanisms. In 2019 35th Symposium on Mass Storage Systems and Technologies (MSST). 65–78. https://doi.org/10.1109/MSST.2019.00-15

Appendix: Artifact Description/Artifact Evaluation

SUMMARY OF THE EXPERIMENTS REPORTED

Please first follow the README in the FT-SZ github link to install the software: FT-SZ. The installation process is the same with the original SZ (https://github.com/szcompressor/SZ). Just remember to use "–enable-randomaccess" options to enable independent block compression when compile. After installation, run some example in the example directory to make sure no error happen.

Experiment in Section 6.2: independent-block compression This section evaluate the compression ratio and PSNR on two datasets (NYX and Hurricane both can be found in Artifact 2: SDR Bench). The curves in Figure 3 are obtained by changing the compression error bounds so that we have different PSNRs with different compression ratio (or bit rates). We repeat the process for different data block sizes. In Figure 4, we evaluate the decompression eficiency when we want to decompress only part of the data. This is set different start/end coordinates you want to decompress. (You can see the help info by running the compiled binary named sz without any arguments so that you know how to pass the arguments to decompress partial data).

Experiment in Section 6.3: compression overhead in compression ratio and compression/decompression rate in error free case. We evaluate based on three versions of SZ. 1. sz: sz without random access enabled; 2. rsz: sz with random access enabled but without fault tolerance; 3. ftrsz: sz with both random access and fault tolerance. Notice ftrsz is this papers contribution. And rsz could be seen as the overhead breakdowns when we only introduce random access. Just run those three versions of sz on the four datasets with the reported error bounds in the paper multiple times (compression ratio is deterministic but run time is non-deterministic. so we run multiple times and take the average). We could get the results reported in table 2 and figure 5.

Experiment in Section 6.4: resilience evaluation with injected errors We inject errors with two modes. One mode is at source code level and the other is at system level. The source code level inject will randomly select an element from input array or quantization bin array. And then it randomly flip a random bit in that element from 1 to 0 or 0 to 1 (one can search "flip" to see the implementation of injection in the file "sz/src/sz_float.c"). We do the same thing for sz and ftrsz multiple times and observe if the program crash or succeed with silent errors. Results can be seen in Table 3. Then we inject error at system level for the whole memory using check-point restart tool. The tool can be found in the paper reference of BLCR (https://crd.lbl.gov/departments/computer-science/class/research/past-projects/BLCR/). The injection is as follows. We profile the general run time of the execution so that we know how long a program will take. Then we use BLCR to stop the program at any random time to dump the memory to file. Then we terminate the program. Next, load the checkpointing file into memory and flip a bit. Then continue the program again with the error-injected checkpoint file. Thus, we observe the error cases: crash or finish with silent errors. Results are reported in Table 3 and figure 6.

Section 6.5: parallel experiment One can follow the README to run our parallel experiment. The readme can be found in https://github.com/sli049/FT-SZ/tree/master/Parallel_experiment. One can also see one of our parallel job script (test_256.job) when running at 256 cores in the same github directory. The Globus simulation code can be found in https://github.com/sli049/FT-SZ/tree/master/Globus_sim_exp.

Author-Created or Modified Artifacts:

```
Persistent ID: https://github.com/sli049/FT-SZ
Artifact name: FT-SZ

Persistent ID: https://sdrbench.github.io/
Artifact name: SDR Bench

Persistent ID: https://github.com/sli049/FT-SZ/blob/
↔  master/Horizon-data.tar.gz
Artifact name: Pluto NASA data
```

BASELINE EXPERIMENTAL SETUP, AND MODIFICATIONS MADE FOR THE PAPER

Relevant hardware details: 36 cores of Intel Xeon E5-2695 v4 processors per node, POSIX I/O, Omni-Path Fabric Interconnect

Operating systems and versions: Operating System: CentOS Linux 7 (Core) CPE OS Name: cpe:/o:centos:centos:7 Kernel: Linux 3.10.0-1127.18.2.el7.x86_64 Architecture: x86-64

Compilers and versions: gcc (GCC) 4.8.5 20150623 (Red Hat 4.8.5-39)

Applications and versions: SZ 2.1

Libraries and versions: Intel(R) MPI Library for Linux* OS, Version 2017 Update 3 Build 20170405 (id: 17193), ZSTD and Zlib versions are included in source code

Key algorithms: SZ 2.1, checksum, ABFT

Input datasets and versions: Scientific data sets can be found in Artifact 2; NASA data can be downloaded with this link: https://pds.nasa.gov/datasearch/subscription-service/SS-Release.shtml.

URL to output from scripts that gathers execution environment information.

https://github.com/sli049/FT-SZ/blob/master/envs.txt