UC Riverside
UC Riverside Electronic Theses and Dissertations

Title
Designing Efficient and Resilient Lossy Compressors for Large-Scale Scientific Computing

Permalink
https://escholarship.org/uc/item/8h24z01x

Author
Li, Sihuan

Publication Date
2020

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA
RIVERSIDE

Designing Efficient and Resilient Lossy Compressors for Large-Scale Scientific Computing

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Sihuan Li

December 2020

Dissertation Committee:

Dr. Zizhong Chen, Chairperson
Dr. Laxmi Bhuyan
Dr. Daniel Wong
Dr. Zhijia Zhao

Copyright by
Sihuan Li
2020

The Dissertation of Sihuan Li is approved:

Committee Chairperson

University of California, Riverside

Acknowledgments

I am very grateful to my advisor, Dr. Zizhong Chen for his advising, encouragement and all support during my more than five years of PhD studies and life. Without him, I would never imagine that I could choose and finish my PhD program in computer science. His insightful ideas on our research projects have contributed a lot to the novelty of my work. Dr. Chen is always accessible when I need some help from him. I really appreciate all the moments when Dr. Chen and me sat down to discuss a tough research problem and after the discussion with him, I always felt like Dr. Chen had thrown lights on my project. Besides being professional, Dr. Chen is also patient and encouraging whenever I encounter some obstacles during my PhD studies. I still remember the warm words from Dr. Chen after I got upset about my paper rejects. His encouragement has been playing an important role when I was having a tough time trying to publish my first piece of my research work. I learnt a lot from him being professional, responsible, accessible, patient, passionate and dedicated which will absolutely benefit me in my future work and life. All in all, I appreciate all the great help from Dr. Chen.

I would like to thank Dr. Laxmi Bhuyan, Dr. Daniel Wong and Dr. Zhijia Zhao for being my dissertation committee members. Their insightful and constructive comments and suggestions have helped me present my dissertation thesis in a much more smooth and structured way.

I feel honored to be mentored by Dr. Franck Cappello and Dr. Sheng Di during my almost two years of internship at Argonne National Laboratory where I conducted a couple of collaboration research projects under their guidance. Dr. Cappello is knowledgeable

and can give very reasonable improvement suggestions on my work. He is also a rigorous researcher from whom I really learnt a lot. Dr. Di is always accessible and easy to work with. Frequent meetings with him have been making my progress move forward smoothly. Dr. Di also helped me greatly with polishing my research writing which undoubtedly improved the overall quality of my work. Therefore, no words can express my appreciation to Dr. Cappello and Dr. Di.

I want to say thanks to all the wonderful lab members in the research group led by Dr. Chen at University of California, Riverside. Thanks to Dr. Li Tan, Dr. Panruo Wu, Dr. Dingwen Tao, Dr. Hongbo Li, Dr. Jieyang Chen, Dr. Xin Liang, Kaiming Ouyang, Yuanlai Liu, Kai Zhao, Yujia Zhai, Quan Fan, Elisabeth Giem and Jinyang Liu. Special thanks to Hongbo Li, Xin Liang and Elisabeth Giem who helped me improve my research paper in a significant amount of detail.

Funding Acknowledgment I appreciate the funding support from National Science Foundation under grant number 1619253, 1513201, from the Exascale Computing Project 17-SC-20-SC (a collaborative effort of two Department of Energy organizations – the Office of Science and the National Nuclear Security Administration) and from the Department of Energy, Office of Science, under contract DE-AC02-06CH11357.

Publication Acknowledgment I acknowledge that part of the thesis has been published or released on line previously.

- Chapter 2 [44] was published in the proceedings of the 2018 IEEE International Conference on Big Data (Big Data), Seattle, Washington, USA, December 10 - 13, 2018.

- Part of chapter 3 [45] was released online in the arXiv library in October, 2020. The other part was published in the proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, Denver, Colorado, USA, November 17 - 22, 2019
- Chapter 4 [46] was published in the proceedings of the 2020 IEEE International Conference on Cluster Computing (CLUSTER), Kobe, Japan, September 14 - 17, 2020.

To my wife, Yiwen Su, and my parents for all the encouragement and support.

ABSTRACT OF THE DISSERTATION

Designing Efficient and Resilient Lossy Compressors for Large-Scale Scientific Computing

by

Sihuan Li

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, December 2020
Dr. Zizhong Chen, Chairperson

Extremely large scale scientific simulation applications have been very important in many scientific domains including cosmology, climate, fluid dynamics, chemistry and so on. It has been shown that running the simulations at a larger scale can bring more discoveries. On one hand, with the increasing scale of those applications, the saturated I/O bandwidth can slow down the execution of the simulation significantly because of the huge amount of data needed to be dumped to the storage system. On the other hand, soft errors striking the simulations are not negligible considering the great number of components in the supercomputer and a single scientific execution spending days to finish. Therefore, it is meaningful to reduce the I/O time and harden the resilience of those large scale simulations. Though hardware solutions like designing new storage systems or error resilient computing devices have great generality, it usually takes longer development time and much more effort than software solutions. This thesis seeks software solutions by designing efficient and resilient lossy compressors for large scale scientific simulations.

To improve the overall simulation performance, we propose a better lossy compressor which has a much higher compression ratio to reduce the I/O time significantly. More specifically, we focus on particle based scientific simulations. As we know, greater compression ratios imply less data to be written to the storage system which in turn, reduces I/O time. The state-of-art lossy compressor takes the advantage of spatial smoothness to achieve high compression ratios. However, particle based simulations have very limited smoothness in space which leads to inadequate compression ratios. In contrast, we propose to exploit smoothness in time for lossy compression and design an optimized compression model based on the existing lossy compressor. Results show our optimized compression model achieves much better compression ratios and significantly reduces I/O time at large scale.

To improve the resilience of the simulation applications equipped with lossy compression, we design soft error resilient schemes for lossy compressors. First, we provide an algorithm-scope protection for one widely used lossy compressor named SZ. Then, we provide an application-scope protection that can be applied to all error-bounded lossy compressors. The algorithm-scope protection can only cover soft errors happening during the execution of the lossy compression itself while the application-scope protection can cover soft errors during simulation, lossy compression and even data writing. Both the algorithm-scope and the application-scope protections can provide significantly better resilience but keep the performance overhead low.

Contents

List of Figures xiii

List of Tables xv

1 Introduction 1

2 Optimizing Lossy Compression with Adjacent Snapshots for N-body Simulation Data 5

2.1 Introduction . . . 5
2.2 Research Background . . . 9
2.3 Problem Formulation . . . 11
2.4 Understanding the N-Body Simulation Data . . . 13
2.4.1 Characteristics of Particles in Consecutive Snapshots . . . 13
2.4.2 HACC variable visualization . . . 14
2.5 Optimized Error-Bounded Compression Model for N-Body Simulation Framework . . . 16
2.6 An Efficient Particle Alignment Mechanism . . . 18
2.7 Optimization of Data Compression . . . 21
2.7.1 Optimizing Data Prediction Accuracy . . . 21
2.7.2 Optimizing Frequencies of Space-Based Compression vs. Time-Based Compression . . . 22
2.8 Performance Evaluation . . . 26
2.8.1 Experimental Setting . . . 26
2.8.2 Evaluation Results . . . 28
2.9 Related Work . . . 33
2.10 Summary . . . 35

3 SDC Resilient Error-bounded Lossy Compressor 37

3.1 Introduction . . . 37
3.2 Background and Problem Formulation . . . 41
3.2.1 SZ Lossy Compression Framework . . . 41
3.2.2 Algorithm based fault tolerance (ABFT) . . . 42

3.2.3 Error model and assumptions 43
3.2.4 Formulation of SDC Detection Evaluation in SZ 43
3.3 Resilience Analysis of SZ 2.1 44
3.3.1 SDC Resiliency – Computation error 45
3.3.2 SDC Resilience – Memory error 49
3.4 Error Tolerance Methodology 51
3.4.1 Blockwise independent design 51
3.4.2 Fault tolerant compression 52
3.4.3 Fault tolerant decompression 54
3.4.4 Avoiding round off errors in checksums 54
3.4.5 Impact to compression ratio without protecting regression and sampling 57
3.5 Discussion for SZ Time Based Compression 57
3.5.1 Introsort 58
3.5.2 Comparison Errors 61
3.5.3 Efficient Error Resilience for Introsort 62
3.6 Experimental Evaluation 62
3.6.1 Experimental Setup 62
3.6.2 Evaluation of Independent-block Compression 65
3.6.3 Error free experimental results 67
3.6.4 Error injected experimental results 69
3.6.5 Parallel experimental results 73
3.7 Related Work 73
3.8 Summary 75

4 Towards End-to-end SDC Detection for HPC Applications Equipped with Lossy Compression 76
4.1 Introduction 76
4.2 Problem formulation 81
4.3 Preliminary Concept and Background 83
4.3.1 Adaptive Impact Driven SDC Detector (AID) 83
4.3.2 Error-bounded Lossy Compression 84
4.4 Data-analytic based End-to-end SDC Detection 85
4.4.1 Design Overview 87
4.4.2 Impact Factor vs. Compression Error Bound 89
4.4.3 Solution A: Synchronous End-to-End SDC Detection with Separate Comparisons (SESD(S)) 92
4.4.4 Solution B: Synchronous End-to-End SDC Detection with Coupled Comparisons (SESD(C)) 95
4.4.5 Solution C: Asynchronous End-to-End SDC Detection 96
4.4.6 Inaccuracy that impacts SDC detection 97
4.4.7 Overhead Analysis 98
4.5 Evaluation and Discussion 100
4.5.1 Experimental Setup 100
4.5.2 Investigation of False Positives in Error-free Cases 100
4.5.3 Investigation of Detection Performance in Erroneous Cases 105

4.5.4 Performance overheads in parallel environment 106
4.6 Related Work 107
4.7 Summary 109

5 Conclusions 110

Bibliography 112

List of Figures

2.1 Particle-overlapping percentage of HACC data 14
2.2 Visualization of HACC variable in space dimension 15
2.3 Visualization of HACC variable in time dimension 15
2.4 Principle of our optimized error-bounded compressor 17
2.5 Optimized in situ compression by combining space-based prediction and prestep-based prediction 18
2.6 Main idea of Algorithm 1 20
2.7 Optimal intervals with different particle-overlapping percentages 25
2.8 Rate distortion of HACC and EXAALT data on variable x 28
2.9 Compression and decompression rate with different error bounds 31
2.10 Parallel compression/decompression and I/O performance. Slashed is SZ_single; dotted is SZ_prestep; solid is SZ_vlct. The stacked bar represents compressing, writing decompressed data, reading decompressed data, and decompressing time from bottom to top. 32

3.1 Analysis of fault tolerance ability for SZ with computation error 45
3.2 Example introsort execution; phase 1 comprises quicksort and heapsort; phase 2 executes insertion sort. 61
3.3 Visualization of Original Data vs. Decompressed Data (Pluto photo taken by New Horizons [6]; SZ compression using Value-range based error bound: 1E-3) 63
3.4 Rate distortion with different block sizes 66
3.5 Efficiency of random access decompression 67
3.6 Compression time and decompression time overheads. Dash lines are random access SZ; solid lines are fault tolerant random access SZ. 69
3.7 Experimental results using evaluation mode B 71
3.8 Compression ratio decrease with cmput. errors 72
3.9 Performance of data dumping/loading (sz vs. ftrsz) 73

4.1 Analysis of time cost in the parallel simulation - FLASH (Sedov) 79
4.2 Parallel simulation with synchronous I/O 82
4.3 Parallel simulation with asynchronous I/O 82

4.4 Design workflow of end-to-end SDC detectors at iteration i 88
4.5 Visual Quality of Reconstructed Data with Different Compression Error Bounds (Climate Simulation CESM: FLDSC) 91
4.6 Distribution of Compression Errors 92
4.7 False positive rate of CESM 102
4.8 False positive rate on Exaalt 102
4.9 False positives on Flash 103
4.10 Overheads and false positives on Nek5000 103
4.11 Execution time in parallel 106

List of Tables

2.1 dataset information . . . 27
2.2 Compression ratio of HACC velocity fields with different pointwise relative error bounds . . . 29

3.1 Basic dataset information . . . 63
3.2 Compression ratio degradation of random-access SZ (rsz) and fault-tolerant random-access SZ (ftrsz) . . . 67
3.3 Percentage of runs whose maximum absolute error is within error bounds in sz and ftrsz . . . 68

4.1 Table of Key Notations . . . 86
4.2 SZ Compression Quality/Ratios of CESM FLDSC . . . 92
4.3 Basic info about scientific applications used in experiments . . . 100
4.4 Acceptable Error Bounds and Compression Ratios (SZ) . . . 104
4.5 Acceptable Error Bounds and Compression Ratios (ZFP) . . . 104
4.6 Confidence Radius of Solution B and C on CESM with SZ at Time Step 40 105

Chapter 1

Introduction

The topics of performance and error resilience are becoming more and more important for large scale scientific simulation applications. Studies have shown that running even larger scale applications is still very critical to more achievements an discoveries [34]. To support the large scale applications, powerful supercomputers have been used to accommodate the demanding computing power and storage requirement. Even with the support of supercomputers, the large scale application still runs in the measurement of days or weeks [39]. With such a long time execution at such a large scale, the supercomputers will spend a significant amount of time in the I/O part because of the generated data to be stored into the disk and they will experience faults. For example, a single cosmological simulation can produce up to 22 PB data which takes more than 10 hours of I/O time even with modern storage systems [34, 51]. For another example, the supercomputer Jaguar has been experiencing double-bit-flip once a day in its memory system [33]. The large portion of I/O time will slow down the execution of the scientific simulations significantly. The faults will

eventually either crash the running process or break the correctness of the running results silently which makes the analysis on this data not trustworthy. The crashes can be resolved by the well studied checkpoint restart techniques [19, 64]. Thus, this thesis mainly focuses on the latter one which is soft error or silent data corruption (SDC). Thus, it is urgent to improve the I/O performance and error resiliency for these large scale scientific simulation applications. Though performance and resilience seems two different topics, resilience is essentially a performance issue. If we do not care about performance, resilience can always be done using redundant operations. For example, the same calculation can be executed twice to see if they are identical or not. Obviously, this method provides resilience with very high performance overhead. So the goal is to provide resilience with low overhead.

To improve the I/O performance, we focus on improving the compression ratio of lossy compressors especially for a specific kind of scientific simulations, the N-body simulations. To improve the error resilience, we first design a soft error resilient lossy compressor based on efficient ABFT (algorithm-based fault tolerance) methods which can protect the algorithm from soft errors. Since lossy compression is being adopted by more and more scientific simulations [34, 39, 60], we believe providing error resilient lossy compression will be meaningful to improve the fault tolerance of the overall scientific simulation applications at large scale. We then extend the protection scope from lossy compression to the scientific simulations that use lossy compression.

The main innovations of the thesis fill several gaps the current literature has. For the compression ratio improvement of lossy compressor on N-body simulation data, the existing work only takes advantage of the characteristics of the data in space dimension

which limits the compression ratio on N-body simulation data since this kind of simulation is based on moving particle instead of a fixed simulation field. However, this thesis proposes new methods to combine the characteristics in both time and space dimension. For the soft error resilient lossy compressor, there was no previous work like us focusing on practical fault tolerance methods which have low run time overheads. Existing redundancy based resilient solutions [9,43] will bring more than 100% overheads in general. For the application-scope resilient solutions for scientific simulations using lossy compression, existing work only considers protecting the simulation part. While our work also considers the compression part and I/O part which basically provides an end-to-end coverage.

We highlight and list the main contributions of the thesis as following.

- We significantly improve the compression ratio of SZ compressor for N-body simulations. The new compression model involves both time and space smoothness. The improved compression ratio greatly reduces the I/O time at large scale with up to 2k cores.

- We are the first to design a soft error resilient version of the lossy compressor, SZ. It can significantly reduce the SDC rate with low overheads in compression ratio, compression speed and decompression speed. Experiments show the overheads can be even smaller at large scale up to 2k cores.

- We are the first to investigate the end-to-end soft error detection for scientific simulations using lossy compression. Our experiments show that existing detectors can still be applied to lossy data with small detection accuracy loss. Also the overheads are tested within 10% at the scale of more than 1k cores.

To summarize, in this chapter, we briefly motivate the topics we work on and the interplay between the topics. Then we present the high level ideas about how our solutions handle each specific problem. After that, we briefly discuss the lack of existing work and novelties of our work. Finally, we highlight the main contributions we make in the thesis. Each of the listed contributions will be presented in more detail later as a single chapter.

Chapter 2

Optimizing Lossy Compression with Adjacent Snapshots for N-body Simulation Data

This chapter improves the overall performance of the N-body simulations by improving the compression ratios of one of the state-of-art lossy compressors, SZ. See how our improved design can reduce the parallel I/O time by up to 20% at large scale of around 2k cores.

2.1 Introduction

N-body simulation represents a significant research category related to moving particles over time, which may produce a vast volume of data for post-analysis. The Hardware/Hybrid Accelerated Cosmology Code (HACC) [34], for example, can simulate trillions

of particles, requiring tens of petabytes of storage space, as indicated by HACC users. In comparison, the storage capacity of the parallel file system (PFS) in Argonne's MIRA Blue Gene/Q system (one of the fastest supercomputers in the world) provides only 20 PB for users to store their data—which cannot meet the storage requirement for even one HACC simulation case, since the PFS storage space is shared with many other users. Significantly reducing the data to store during the simulation is critical to the success of N-body research projects.

Although many state-of-the-art data compressors exist, they cannot significantly reduce the N-body simulation data size. N-body simulation data that need to be dumped are mainly floating-point values, which cannot be compressed effectively by lossless compressors such as Gzip [5], BlosC [3], and FPC [20]. The reason is that most lossless compressors rely on exactly repeated patterns appearing in the stream of bytes, while the stream of floating-point data has few repeated patterns because of random ending mantissa bits on each data point. On the other hand, existing error-bounded lossy data compressors (such as SZ [29, 30, 73], ZFP [53], FPZIP [55], and ISABELA [42]) cannot work effectively on the N-body simulation data because they significantly rely on the consecutiveness of the data in space, while the adjacent data points in one snapshot are often not correlated with each other in space, as will be demonstrated later.

A straightforward idea to significantly improve the data compression quality for the N-body simulation framework is leveraging the possible smoothness of the data in the time dimension. This idea, however, faces some serious issues. First, multiple-snapshot-based trajectory data compression is not suitable because collecting many snapshots on line

before doing the compression is impossible for extremely large-scale simulations, considering the vast data to produce versus limited memory capacity. Second, a large-scale parallel simulation such as HACC involves hundreds of thousands of ranks, each handling a specific region in space. Particles are always moving during the simulation, such that the same rank across adjacent time steps may have different orders of particles. The consecutive temporal snapshots of the same rank may even have varied numbers of particles, such that aligning the particles across the consecutive snapshots is nontrivial. Third, we have to adopt space-based compression (or snapshot-based compression) from time to time during the entire simulation; otherwise, reconstructing/decompressing any snapshots (even near the end of the simulation) would have to decompress all its previous snapshots, causing undesired decompression overhead. How to optimize the overall compression quality by combining the space-based compression and time-based compression needs to be studied carefully.

In this work, we propose a novel, error-bounded lossy compressor for the N-body simulation framework. Our compressor can substantially reduce the size of the storage data during the simulation and can also improve the I/O performance significantly. Our contributions are as follows.

- We propose an efficient error-bounded lossy compression model for N-body simulation data by adopting both a space-based strategy and time-based strategy alternately, with the required information limited to only the current snapshot and the previous snapshot. The compression model can be extended to other scientific simulations if the simulation data exhibit a certain consecutiveness in the time dimension. We also prove the most appropriate frequency for the space-based vs. time-based compression.

- We propose a lightweight particle alignment mechanism, which is a fundamental step to the following time-based compression step for the N-body simulation. The particle alignment approach involves two critical steps, radix-sorting particles and classifying missing particles, which are designed and implemented carefully considering both performance and memory cost.

- We thoroughly evaluate the compression quality of our proposed solution using a cosmological simulation code (HACC) and a molecular dynamics simulation code (EXAALT), which are both widely used in the N-body research community. Experiments show that our solution can significantly improve the compression quality compared with that of all other existing state-of-the-art lossy compressors, including SZ [29, 30, 73], ZFP [53], FPZIP [55], NUMARCK [24], and decimation approaches, with comparable compression/decompression rate. The compression ratio is improved by up to 43% on the velocity field and up to 300% on the position field of HACC data; on EXAALT data, it is improved by up to 260% compared with that of the second-best compressor with the same data distortion level. The parallel I/O performance is improved by up to 20%.

The remainder of the chapter is organized as follows. In Section 2.2 we introduce the research background. In Section 2.3 we formulate the research problem by discussing the target and conditions of the lossy compression work based on our communication with N-body simulation code developers and users. In Section 2.5 we describe our proposed error-bounded lossy compression model. In Section 2.6 we propose an efficient particle alignment mechanism and discuss how to improve the data prediction accuracy in the context of N-

body simulation. In Section 2.8 we present an in-depth analysis and compare our evaluation results with other related work. In Section 2.9 we discuss the related work and in Section 2.10 we summarize our conclusions.

2.2 Research Background

We describe the background of N-body research (including both cosmology and molecular dynamics) in this section. In the absence of analytical methods for cosmological research, numerical simulations are the only methods available for the study of extremely large structures such as galaxies and clusters of galaxies. In the past two decades, many relative N-body simulation techniques have been developed and executed on powerful supercomputers, and the simulation results have provided valuable insights for the study of structure formation. As indicated in the survey [13], cosmological simulations are generally based on an initial condition setup using a power spectrum, and then the particles' positions and velocities are calculated iteratively in each time step until a completion condition is met. Molecular dynamics (MD) simulation plays an important role in discovery of interesting behaviors of atoms [63]. The EXAALT project, for example, aims to develop MD simulation at exascale to address the key fusion and fission energy materials challenges [4]. EXAALT can simulate behavior of atoms in a nano-crystalline sample of copper under the influence of a strong electric field. It will generate a large amount of long trajectory of atoms that requires recording the positions of atoms in each time step.

For an N-body simulation, the positions and velocities of the simulated particles generally compose the critical information to be stored into the PFS for post-analysis. In

HACC simulations [34], for example, each snapshot stored in the PFS contains six fields, x, y, z, v_x, v_y, and v_z, each being a 1D array with exactly the same number of elements (i.e., the number of particles). The first three fields (x, y, z) indicate the particles' positions, and the other three fields (v_x, v_y, v_z) refer to the particles' velocities. Since in one simulation run, there could be hundreds of snapshots each containing trillions of particles, the total data size could easily reach 10 PB or more. The users often adopt decimation in time (storing only one snapshot every several time steps) for the post-analysis.

For a parallel simulation code, there are many ranks to be executed on hundreds of thousands of processors in parallel, and each rank is handling a particular region in the space. Every rank needs to communicate with a few other ranks because the particles may move from one region to another during the simulation. Hence, the same rank across consecutive time steps (or snapshots) may have different orders or even different numbers of particles. In order to trace the particles across snapshots, each particle has an ID number, which is stored in an extra integer array at runtime. The HACC code, for instance, adopts a 64-bit ID array to identify the particles. According to HACC users, post-analysis does not require knowing the ID array, so this information does not need to be stored. Moreover, since the users focus mainly on the characteristics of the spatial structures of the particles' clusters, they do not care about overall orders of the particles, as long as the elements in the six fields (x, y, z, v_x, v_y, v_z) are kept consistent.

2.3 Problem Formulation

In this section, we formulate the research problem by clarifying the conditions and target in our compression work, based on our close communication with N-body researchers.

During an N-body simulation such as a cosmological simulation, we need to compress the particles' information (i.e., each particle's position and velocity) in order to reduce the storage burden as much as possible. Specifically, given simulation with T time steps (or snapshots), we need to design and implement an in situ compressor that can get as high a compression ratio as possible in the compression of all six 1D arrays (x, y, z, v_x, v_y, and v_z) based on the user's error-controlling demands. The compression ratio (denoted by r) is defined as the ratio of the original data size to the compressed data size, as presented in Formula (2.1),

$$r = \frac{(\sum_{i=1}^{T} 6N_i) \cdot data_type_size}{\sum_{i=1}^{T} \sum_{field=x,y,z,v_x,v_y,v_z}^{6} C_i(field)}, \tag{2.1}$$

where *data_type_size* refers to the number of bytes used to represent one data point in the original dataset (4 for single precision and 8 for double precision), N_i means the number of particles at time step i, and C_i(*field*) means the compressed size (in bytes) at time step i for a particular field (selected in $\{x, y, z, v_x, v_y, v_z\}$).

This objective is subject to the following conditions.

- The compression algorithm should have a low time complexity, such that the compression/decompression time would be comparable to that of the existing fast compressors (e.g., SZ [73] and ZFP [53]).

- The compression method should have relatively low memory cost, considering the huge memory footprint consumed. Specifically, only one previous consecutive snapshot is allowed to be used for the compression of the current snapshot in the simulation.

- According to N-body researchers, the position fields (i.e., x, y, z) need to use an absolute error bound, which is a constant number specified by the user to bound the maximum difference between the original data values and decompressed data values. Specifically, given a constant error bound denoted by eb (e.g., eb=1E-4 of the global value range), for each original data point d_i and its corresponding decompressed data point d'_i, the compression errors e_i should follow the following inequality: $e_i=|d_i-d'_i| \leq eb\cdot(d_{max}-d_{min})$, where d_{max}, d_{min} is the maximum and minimum of original data. The eb is also called value-range-based error bound.

- The velocity fields (v_x, v_y, and v_z) should use a pointwise relative error bound, which means that the ratio of the value difference to the original value should be no greater than a user-specified constant number (such as 1%), in that the faster a particle moves, the higher the compression errors it can tolerate intuitively. In this sense, the uniform absolute error bound is not suitable for the compression of velocity fields because (1) a large uniform absolute error bound will lead low-speed particles to be largely skewed from their real states and (2) a relatively small uniform absolute error bound will cause an over-reservation of the precision for the particles with fast speeds, leading to the low compression ratios.

2.4 Understanding the N-Body Simulation Data

In this section, we investigate the characteristics of the N-body simulation data, which are fundamental to the design principle of our compression model.

2.4.1 Characteristics of Particles in Consecutive Snapshots

To understand whether the time-based compression strategy is promising for N-body simulation data compression, we need to investigate the percentage of particles overlapped across two consecutive snapshots, in that a greater portions of overlapped particles will contribute to more significant improvement of compression ratio. Figure 2.1 presents the particle-overlapping percentage based on HACC data using 100 snapshots. Without loss of generality, we assume that the space-based compression is executed every 10 time steps: that is, in a period with 10 consecutive time steps, the first snapshot adopts space-based compression and all others adopt time-based compression with overlapped particles.

In Figure 2.1, the green solid curve refers to the percentage of the overlapped particles between two adjacent snapshots. The red dashed curve shows the percentage of the overlapped particles between the current snapshot and the initial snapshot in that 10-consecutive-time-step period. We summarize two critical insights based on our analysis of our lossy compression design and optimization strategies.

1. The particle-overlapping percentage is high (≥95% every few time steps). This observation indicates that it is feasible to extract overlapped particles across adjacent snapshots and improve the compression ratio by leveraging the data smoothness in the time dimension.

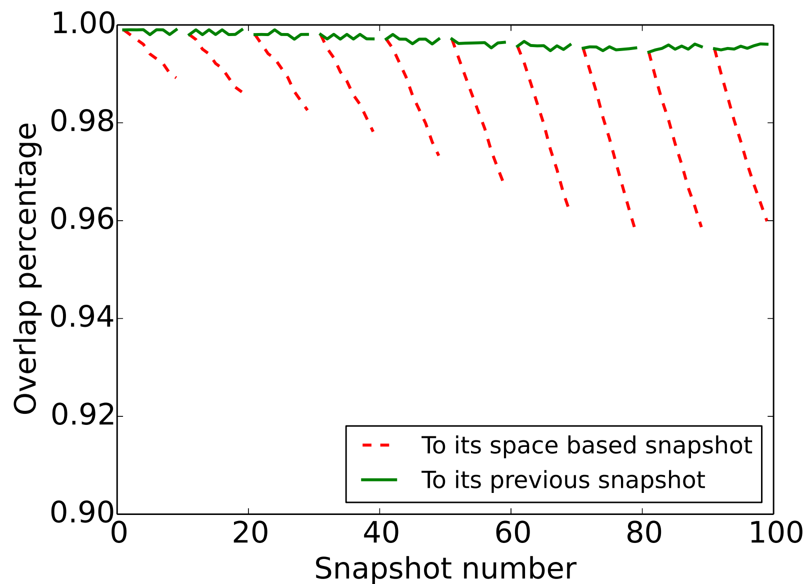
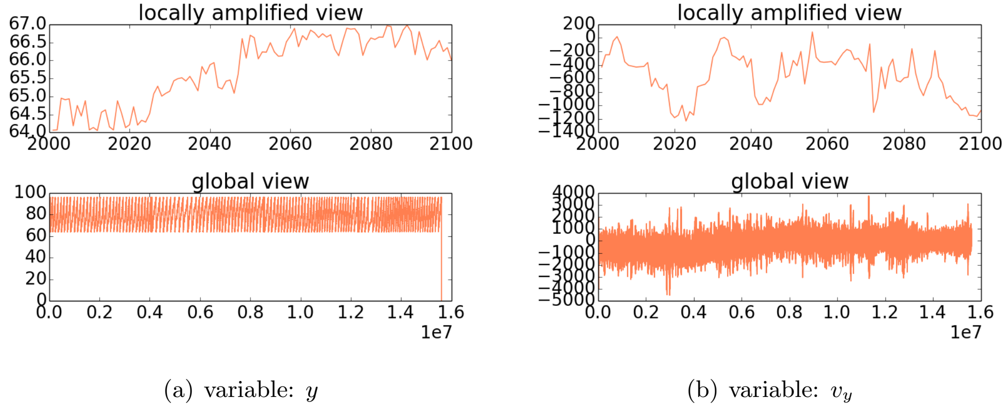


Figure 2.1: Particle-overlapping percentage of HACC data

2. In each period, the particle-overlapping percentage between two adjacent snapshots can be treated as a constant, because of the horizontal green curves observed in the figure. We derive an optimal frequency for space-based compression versus time-based compression, to be described in Section 2.7.2 in detail.

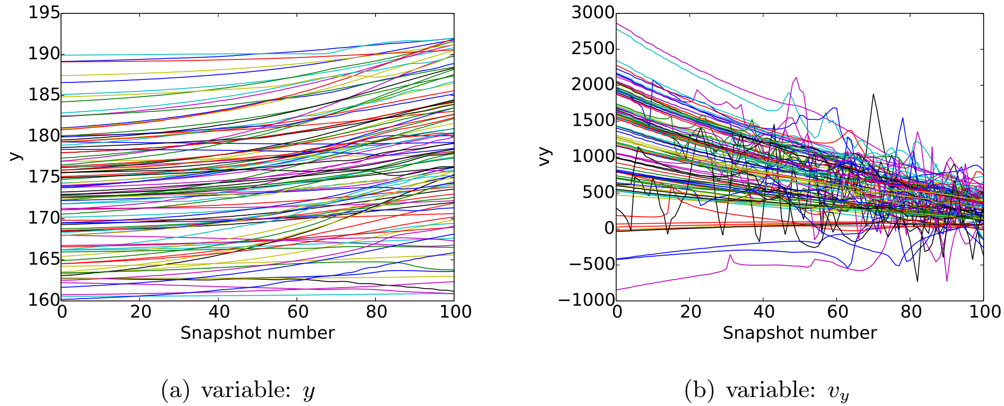
2.4.2 HACC variable visualization

Figure 2.2 presents the HACC simulation data in a snapshot (i.e., in the space dimension). We can clearly see that the spatial consecutiveness of the data is hardly observed especially for velocity fields, which means that the single-snapshot-based compression method would definitely suffer from very limited compression ratios. As shown in Tao et al.'s paper [72], the best compression ratios for the position field and velocity field are only



(a) variable: y (b) variable: v_y

Figure 2.2: Visualization of HACC variable in space dimension



(a) variable: y (b) variable: v_y

Figure 2.3: Visualization of HACC variable in time dimension

about 7 : 1 and 4 : 1, respectively, under the value-range-based error bound of 0.0001 [72] which is not adequate for real world usage.

Figure 2.3 presents the smoothness of the position and velocity values of the same particles simulated over multiple time steps. We randomly chose 100 particles and recorded their position and velocity in time dimension. We observe that the data values are fairly smooth across adjacent time steps, especially for the position field. As for the velocity field, the data exhibits much higher smoothness in time dimension than in space.

2.5 Optimized Error-Bounded Compression Model for N-Body Simulation Framework

In this section, we propose an optimized error-bounded compression model that is particularly effective for N-body simulation data. The design principle is based on the SZ compression model [73], which includes four critical steps: data prediction, linear-scaling quantization, variable-length encoding, and lossless compression. The key difference between our model and the SZ model is twofold: (1) unlike SZ, we design an efficient particle alignment algorithm because we need to leverage the smoothness of the data in the time dimension across consecutive time steps (to be detailed in Section 2.6); and (2): we also improve the prediction accuracy significantly by leveraging more advanced approaches rather than using only space-based prediction methods (to be detailed in Section 2.7.1). We present the whole compression procedure in Figure 2.4. The first two steps are the critical steps in the whole compression, directly determining the final compression quality. In general, the higher the prediction accuracy, the higher the compression quality because of more duplicated quantization codes to be introduced in the linear-scaling quantization step.

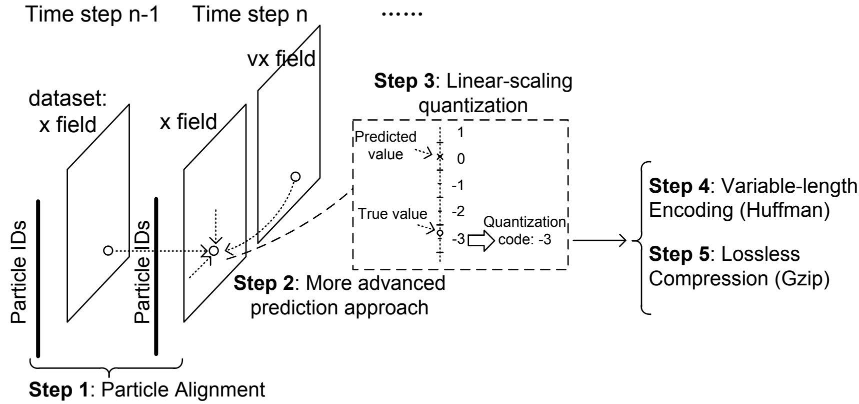


Figure 2.4: Principle of our optimized error-bounded compressor

In Figure 2.5, we illustrate how we combine different types of prediction approaches (or compression techniques). At the beginning, we have to adopt the space-based compression at the first time step because it has no previous time steps. For any following time steps, we can adopt either space-based compression or prestep-based compression because the information at previous time steps would be available to use in the prediction. In fact, we still have to perform one space-based compression every few time steps, such that the data decompression at any following time step would not reply on too many previous time steps. However, how to optimize the frequency of the space-based compression and time-based compression is nontrivial. It may involve multiple factors, including particle-overlapping percentage across consecutive time steps, space-based compression ratio, and time-based compression ratio (to be detailed in Section 2.7.2).

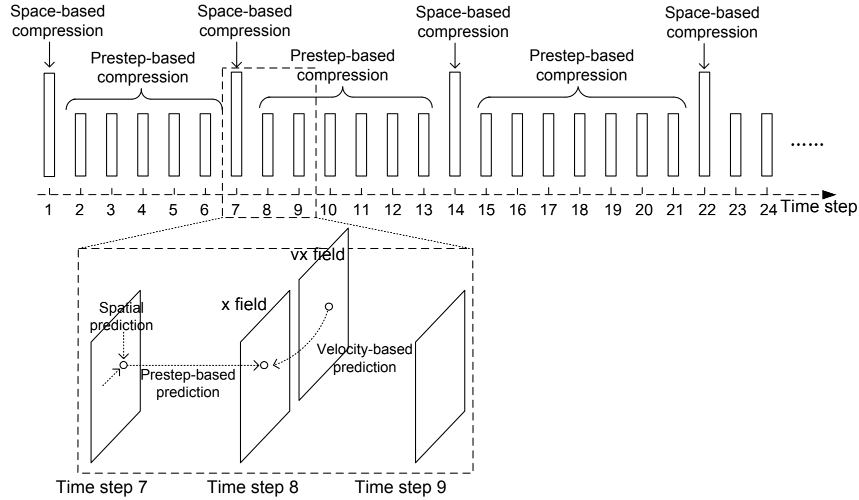


Figure 2.5: Optimized in situ compression by combining space-based prediction and prestep-based prediction

2.6 An Efficient Particle Alignment Mechanism

In this section, we present an on-line algorithm (Algorithm 1) that can align particles across adjacent time steps efficiently, so that we can use prestep-based compression for aligned particles and space-based compression for unaligned ones. On-line particle alignment is nontrivial because (1) for the sake of limited memory cost, at most one previous snapshot is available to use every time the particles of a snapshot need to be aligned; and (2) the algorithm has to be of linear time complexity, since the state-of-the-art compression algorithms generally have linear time complexity. The key idea is using a lightweight, bit-mask array (also called alignment bit-array in the following text) to trace/map the overlapped particles between two consecutive snapshots.

In Algorithm 1, we use a vector $perm[]$ to record the original locations of particles in the dataset. After performing the radix sorting (line 4), we reorganize the data to make the particles' locations consistent with the sorted IDs (line 5$\sim$12). Such a design can effectively

Algorithm 1 On-line particle alignment

Input: particle ID array of the last snapshot (denoted $id_0[]$), the number (denoted l_0) of aligned particles in $id_0[]$, particle ID array of the current snapshot (denoted $id_1[]$), and data of current time step denoted by $var_1[][]$ ($var_1[j]$ represent jth field).

Output: updated ID array ($id_1[]$), data of the fields ($var_1[][]$), the number of aligned particles in current snapshot (l_1), alignment bit-array ($align[]$).

```
1:  for i = 0 → id_1[].length−1 do
2:      perm[i] ← i. /*Initialize the permutation for sorting*/
3:  end for
4:  in_place_radixsort(id_1[], perm[]). /*perm[] records original locations*/
5:  for i = 0 → id_1[].length−1 do
6:      while i ≠ perm[i] do
7:          swap(perm[i], perm[perm[i]]).
8:          for var_1[m] do
9:              swap(var_1[m][i], var_1[m][perm[i]]).
10:         end for
11:     end while
12: end for
13: i, j, k, align[] ← 0. /*for align[], 0 indicates matched particle*/
14: while i < l_0 & j < id_1[].length do
15:     if id_0[i] == id_1[j] then
16:         swap(id_1[k], id_1[j]).
17:         for var_1[m] do
18:             swap(var_1[m][k], var_1[m][j]).
19:         end for
20:         i++, j++, k++. /*counter increment*/
21:     end if
22:     if id_0[i] < id_1[j] then
23:         align[i] = 1, /*1 indicates missing/unmatched particle*/
24:         i++.
25:     end if
26:     if id_0[i] > id_1[j] then
27:         j++. /*keep searching for matched particles*/
28:     end if
29: end while
30: l_1 ← k. /*output the number of aligned particles*/
```

avoid too many memory accesses and swap operations for $var_1[][]$ during the sorting. Then (line 14∼29) we scan the two sorted IDs (id_0 and id_1) to align particles (note that id_0 have been sorted in the last time step) between the two snapshots. Figure 2.6 uses an example to illustrate how Algorithm 1 works.

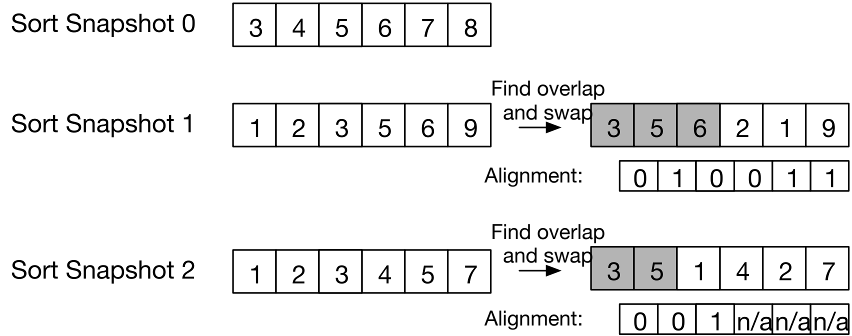


Figure 2.6: Main idea of Algorithm 1

Suppose snapshot 0 is already processed in the last time step and snapshot 1 has been sorted in the current algorithm (line 4). Scanning the two snapshots with two separate pointers, we can move all the unmatched/missing particles (2,1,9) to the end of snapshot 1 and use an alignment bit-array {0,1,0,0,1,1} to record the matched particles' locations in snapshot 0. Note that we use "0" to indicate a matched particle. Similarly, for the sorted snapshot 2 {1,2,3,4,5,7}, we compare them with the aligned particles in snapshot 1 (i.e., 3,5,6) and mark the matched particles 3,5 in snapshot 1 using the alignment bit-array {0,0,1,n/a,n/a,n/a}.

Complexity: Denote the number of fields by N_f, number of particles by n. Consider the complexity for each snapshot. In-place-radix-sort will take $2\frac{b}{l_r}n$ (line 4), where b is the bit length of an original data point; l_r is the radix bit length, and 2 is because the permutation array $perm[]$ is being reordered when $id_1[]$ is being sorted. Reordering the

variables by the permutation of radix sorting takes $N_f n$ (line 5 to 12). Finding the overlapped index of id_0, id_1 and swapping them to former part of id_1 takes at most $(2 + N_f)n$ (line 12 to 24). In total, that is $2(b/l_r + N_f + 1)n$

For memory, we consider only the extra memory usage, not including the memory that the original simulation will take. To do time-based compression, one previous snapshot data is needed, which first doubles the memory usage $(N_f + 1)n$, where 1 is because of the ID array. In addition, the permutation array ($perm[]$) takes n units of extra memory. The alignment bit-array ($align[]$) takes at most n. Since we can free the permutation array after we reorder the variable before the alignment bit-array is initialized, the extra memory usage is just n. So in total, that is $(N_f + 1)n + n = (N_f + 2)n$ units of extra memory.

2.7 Optimization of Data Compression

In this section, we describe how to optimize the compression quality for the N-body simulation data by leveraging its data properties.

2.7.1 Optimizing Data Prediction Accuracy

The HACC application needs to store both velocity and position information during the simulation. We now detail how we improve the prediction accuracy by incorporating them together. Our key idea is to leverage Newton's law: $\vec{x_1} = \vec{x_0} + \vec{v}\Delta t$, where $\vec{x_0}$ is the position of all particles in last snapshot; $\vec{v}$ is the average velocity; $\vec{x_1}$ is the position in the current snapshot; and Δt is just the time difference between current snapshot and last one. The unknown parameters are $\vec{v}$ and Δt. We estimate $\vec{v}$ by $\frac{\vec{v_0}+\vec{v_1}}{2}$, where $\vec{v_0}$ and $\vec{v_1}$

refer to the velocity values of the last and current snapshot, respectively. We estimate the average velocity by assuming a linear motion with constant acceleration. The Δt is calculated based on the minimization of the prediction MSE (mean square error) in the domain $\Delta t \in (-\infty, +\infty)$. The optimization problem is formulated as follows.

$$\min_{\Delta t} \quad mse(\Delta t) \triangleq \frac{1}{n}(\vec{x_1} - \vec{x_0} - \vec{v}\Delta t)^2 \tag{2.2a}$$

$$\text{subject to} \quad \Delta t \in (-\infty, +\infty) \tag{2.2b}$$

$$\Delta t \in \tag{2.2c}$$

Equation (2.2) can be rewritten as follows: $mse(\Delta t) = \frac{1}{n}[\vec{v}^2\Delta t^2 - 2\vec{v}(\vec{x_1} - \vec{x_0})\Delta t + (\vec{x_1} - \vec{x_0})^2]$ where n is the number of common particles between these two adjacent snapshots. The single variable quadratic function can be minimized when $\Delta t = \frac{\vec{v}(\vec{x_1} - \vec{x_0})}{\vec{v}^2} = \frac{(\vec{v_0} + \vec{v_1})(\vec{x_1} - \vec{x_0})}{2\vec{v}^2}$, which implies that the complexity of calculating Δt is $O(n)$.

2.7.2 Optimizing Frequencies of Space-Based Compression vs. Time-Based Compression

Since fewer and fewer overlapped particles occur with the increasing snapshot number, using time-based compression is not always better. We need to do a space-based compression every few time steps. We assume that the particle-overlapping percentage with prestep snapshot is α ($0<\alpha<1$), without loss of generality (see Figure 2.1). Suppose there are totally T snapshots and every space-based compressed snapshot is followed by $k-1$ time-based compressed snapshots, then we can derive the following theorem.

Theorem 1 *In order to minimize the overall compressed file size across T snapshots, the optimal interval k of the space-based compression is either $\lceil k_0 \rceil$ or $\lfloor k_0 \rfloor$ such that $\alpha^{k_0-1}(k_0 \ln \alpha - 1) + 1 = 0$.*

Proof. We assume the space-based compression ratio and time-based compression ratio are two constants for simplicity, and their reciprocals are denoted as r_s and r_t, respectively ($0 < r_s, r_t < 1$).

For this optimization problem, we need to calculate the objective function, with snapshots labeled from 0 to $T-1$. We denote the original data size of each snapshot as 1 without loss of generality, then we can calculate the compressed file sizes for the snapshot 0 through $k-1$ as follows. Snapshot 0 will be compressed to $1 * r_s$ since it is compressed only by space-based compression. The snapshot i ($i \in [1, k-1]$) will be compressed to $1 \cdot \alpha^i \cdot r_t + 1 \cdot (1-\alpha^i) \cdot r_s$ since the overlapped part $1 * \alpha^i$ will be compressed by time-based compression and the remaining part by space-based compression. Note that the analysis is the same for snapshots k through $2k-1$; $2k$ through $3k-1$, and so on. Without loss of generality, we assume T is divisible by k, so the overall compressed file size is $\frac{T}{k}[r_s + \sum_{i=1}^{k-1} \alpha^i r_t + (1-\alpha^i) r_s]$. The inner part is just a $k-1$ term geometric sequence, thus we can simplify the function as follows.

$$f(k) \triangleq T[r_s + \frac{\alpha}{1-\alpha}(r_t - r_s)\frac{1-\alpha^{k-1}}{k}] \tag{2.3}$$

$$\min_{k} \quad f(k) \tag{2.4a}$$

$$\text{subject to} \quad k \in [1, +\infty) \tag{2.4b}$$

$$k \in . \tag{2.4c}$$

The optimization problem can be formalized using equation 2.4. Now, we just need to prove that the objective function $f(k)$ in the domain $k \in [1, \infty)$ has one and only one minimum. Taking the first-order derivative leads to $f'(k) = \frac{T\alpha(r_s - r_t)}{1-\alpha} * \frac{1}{k^2} * [\alpha^{k-1}(k \ln \alpha - 1) + 1]$. The first factor $\frac{T\alpha(r_s - r_t)}{1-\alpha}$ is always positive in the case where time-based compression has a better compression ratio than does space-based compression ($r_s > r_t$). Similarly, the second factor is also positive. Thus, whether $f(k)$ will increase or decrease with increasing k depends on the sign of the third factor $g(k) \triangleq \alpha^{k-1}(k \ln \alpha - 1) + 1$. What we can ensure first is that $g(1) = \ln \alpha < 0$ and $\lim_{k\to\infty} g(k) = 1 + \lim_{k\to\infty} \alpha^{k-1}(k \ln \alpha - 1) = 1 > 0$. Since $g(k)$ is continuous in domain $k \in [1, \infty)$, $g(k)$ has at least one root. Then we prove $g(k)$ has only one root. Taking the first-order derivative leads to $g'(k) = \ln^2 \alpha * k\alpha^{k-1} > 0 \ \forall k \in [1, \infty)$. That is, $g(k)$ is monotonically increasing in $[1, \infty)$. Recalling that $g(1) < 0$ and $\lim_{k\to\infty} g(k) > 0$, we conclude that in $[1, \infty)$, $g(k)$ has one and only one root denoted by k_0 such that $g(k_0) = 0$. So $f(k_0)$ will be a global minimum in $[1, \infty)$.

This above analysis assumes k is a real number, while it is actually an integer. So the optimal k should be either $\lceil k_0 \rceil$ or $\lfloor k_0 \rfloor$, where $g(k_0) = \alpha^{k_0 - 1}(k_0 \ln \alpha - 1) + 1 = 0$. ■

To give some idea of how large the optimal interval is with different overlapping percentage, we plotted Figure 2.7 using different overlapping percentages in $[0.01, 0.99]$ with increasing step 0.01.

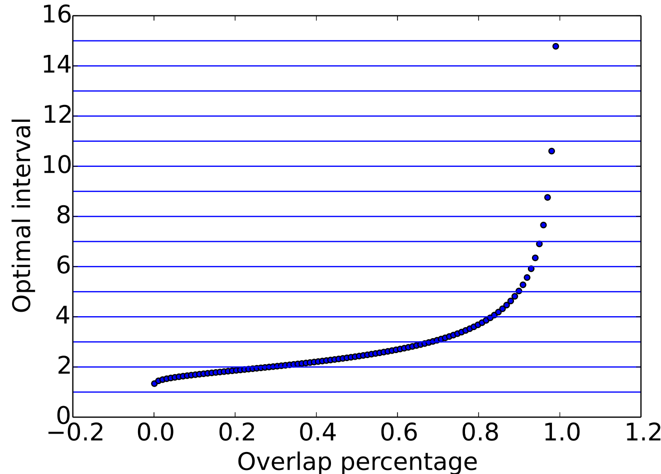


Figure 2.7: Optimal intervals with different particle-overlapping percentages

Algorithm 2 details how to calculate the optimal interval. Notice that the overlapping percentage should be calculated before calling Algorithm 2. At runtime, we just estimate it using the number of aligned particles in the next snapshot following a pure space-based compressed snapshot divided by the number of particles in that space-based compressed snapshot. Notice that a greater optimal interval means more time to get a specific decompressed snapshot for the users. We allow a user defined maximum interval i_{max} to be a parameter. We first test whether $g(k)$ at 1 and i_{max} have the same sign. If so, there is no root of $g(k)$ in $[1, i_{max}]$, implying i_{max} is the optimal interval. Otherwise, one and only one root is guaranteed in $[1, i_{max}]$. We solve it using binary search. Then check two closest integers and select the one with lower $f(k)$ (Formula (2.3)) as the final solution.

Complexity: The dominant part of Algorithm 2's complexity is the binary search. Since we need to identify the closest integer solution, we can terminate the search as the current search interval is less than 1. Thus, our binary search part costs $\log i_{max}$ times of function evaluations.

Algorithm 2 OPTIMAL INTERVAL SELECTION

Input: an on-line computed overlapping percentage α, a user-defined interval threshold i_{max} ($i_{max} = 10$ by default)

Output: optimal interval k

1: $g(k) \leftarrow \alpha^{k-1}(k \ln \alpha - 1) + 1$
2: **if** $g(1) * g(i_{max}) > 0$ **then**
3: $\quad k = i_{max}$ /*no root in $[1, i_{max}]$*/
4: **else**
5: $\quad k =$ binary search root of $g(k)$ in $[1, i_{max}]$
6: $\quad h(k) \leftarrow \frac{1-\alpha^{k-1}}{k}$
7: $\quad$ **if** $h(\lfloor k \rfloor) > h(\lceil k \rceil)$ **then**
8: $\quad\quad k \leftarrow \lfloor k \rfloor$ /*compare which integer solution is better*/
9: $\quad$ **else**
10: $\quad\quad k \leftarrow \lfloor k \rfloor + 1$
11: $\quad$ **end if**
12: **end if**

2.8 Performance Evaluation

2.8.1 Experimental Setting

We conduct our experimental evaluations on a supercomputer [1] using up to 2,048 cores (i.e., 128 nodes, each with two Intel Xeon E5-2695 v4 processors and 128 GB of memory, and each processor with 16 cores). The I/O and storage systems are typical high-end supercomputer facilities. We use the file-per-process mode with POSIX I/O [81] on each process for reading/writing data in parallel.

We test our proposed compression framework with optimized techniques using two particle simulation datasets. One is the cosmological simulation dataset HACC, and the other is the molecular dynamics simulation dataset EXAALT. The basic information about these two datasets is listed in Table 2.1. Note that these datasets are from real simulations and their structures and properties are similar to those of much larger datasets from extreme-scale simulations (trillions particles cosmology simulation).

We mainly focus on two metrics to assess the compression quality. The first metric is compression ratio based on the same pointwise relative error bound, which is mainly used to evaluate velocity fields. The other important evaluation metric is peak signal-to-noise ratio (PSNR), as defined in Formula (2.5). PSNR is commonly used to assess the overall distortion between the original data and reconstructed data especially in visualization. Logically, the higher the PSNR is, the lower the overall compression error, indicating a better compression quality.

$$psnr = 20 \cdot log_{10}(\frac{d_{max} - d_{min}}{rmse}) \tag{2.5}$$

where $rmse = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(d_i - d_i')^2}$, and d_{max} and d_{min} refer to the max and min value respectively. We will present the rate-distortion figure, which shows the bit-rate (the number of bits used as per data point after the compression) versus the PSNR value.

Table 2.1: dataset information

	Snapshots	Variables	Particles (million)	Total Size
HACC	100	x, y, z, v_x, v_y, v_z	15-20	36GB
EXAALT	83	x, y, z	1	1GB

2.8.2 Evaluation Results

Alignment bit-array overhead

We first demonstrate that the alignment bit-array in our Algorithm 1 has negligible storage overhead. The alignment bit-array does not change with various error bounds. The overall compressed alignment bit-array occupies only 15.70 MB, which is far less than the original HACC data size (36 GB). Such an overhead is negligible unless the compression ratio is up to 1000X or so, which is not the case for HACC.

Rate distortion

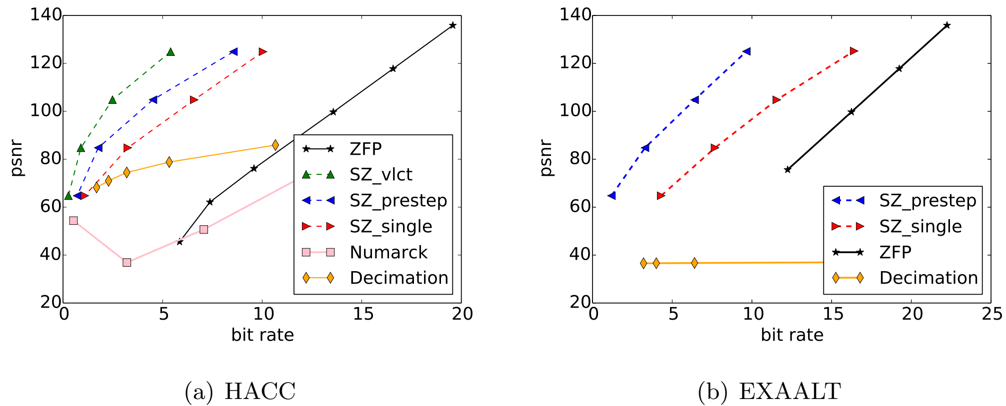


Figure 2.8: Rate distortion of HACC and EXAALT data on variable x

For the HACC data, we evaluate our proposed methods (SZ_vlct, SZ_prestep) by comparing to other existing compressors shown in Figure 2.8. SZ_vlct adopts all the optimization strategies we proposed above (including Section 2.6 and Section 2.7). Unlike SZ_vlct, SZ_prestep adopts only prestep data values to predict the data value in the current time step, without leveraging Newton's law. The results are obtained by setting the value-

range-based error bound to 1E-3 $\sim$ 1E-6 for SZ-based methods. In the SZ_vlct solution, we always use the pointwise relative error bound 0.1 for the velocity to predict positions. The decimation method uses linear interpolation. As we can see, for the HACC position data, our proposed methods (SZ_vlct, SZ_prestep) outperform the second-best method, the single-snapshot-based SZ (SZ_single), by 14% and 46% in bit-rate given the fixed PSNR at around 124. The 46% improvement in bit-rate reflects around 85% improvement in the compression ratio. The improvement is more significant if users accept lower PSNR. For example, SZ_vlct has 4X compression ratio of SZ_single with the same PSNR of about 64. For the EXAALT data, we omit Numarck because of its low compression ratio. We cannot adopt SZ_vlct for EXAALT data because it has only position fields. Also, we cannot use FPZIP because it does not support value-range-based error bound. Evaluation results show that our proposed method (SZ_prestep) outperforms the second-best (SZ_single) method by 40% in bit rate given the fixed PSNR at around 125. Again, the improvement is more significant if users accept lower PSNR. For example, SZ_prestep has 3.6X compression ratio of SZ_single at PSNR 64.

Evaluation for pointwise relative error bound

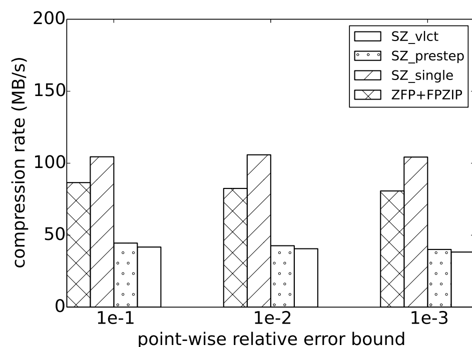
Table 2.2: Compression ratio of HACC velocity fields with different pointwise relative error bounds

	1E-1			1E-2			1E-3		
	v_x	v_y	v_z	v_x	v_y	v_z	v_x	v_y	v_z
SZ_vlct	13.90	12.96	11.60	7.59	7.22	6.69	4.64	4.45	4.19
Numarck	11.28	10.78	10.08	5.69	5.49	5.21	2.97	2.92	2.91
SZ_single	9.33	8.66	7.42	4.91	4.69	4.24	3.24	3.14	2.93
FPZIP	5.67	5.33	4.98	3.71	3.56	3.40	2.75	2.66	2.58

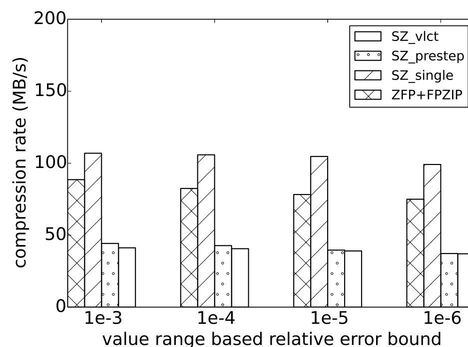
This section evaluation part involves only HACC data since EXAALT data has no velocity field. Since decimation and ZFP do not support pointwise relative error bounds, we exclude them in this evaluation. The results are shown in Table 2.2. It is observed that the SZ_vlct has 15+% compression ratio improvement compared with the second-best method. The improvement is up to 43% when the error bound is about 0.001, because of its much higher prediction accuracy. Although our method has similar compression ratios with Numarck when the error bound is set to 0.1, Numarck actually is a worse solution because it cannot respect the pointwise relative error bounds. Specifically, when setting the pointwise relative error to 1E-3, Numarck's actual maximum compression error would reach up to 1504 in our experiment.

Compression and decompression rate

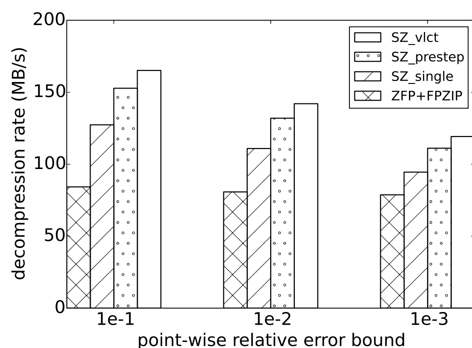
In Figure 2.9, We present the compression and decompression rate of our proposed SZ_vlct and SZ_prestep on the HACC and EXAALT data. We can see that for HACC, our compressor's compression rate is about $\frac{1}{3}$ as high as that of SZ_single and 50% as high as that of ZFP+FPZIP, because of the particle alignment cost. For the decompression rate, our proposed compressor runs much faster than both SZ_single and ZFP+FPZIP, in that the time-based method has no particle alignment cost.Note that SZ_vlct is even faster than SZ_prestep in decompression. The reason is that SZ_vlct has higher prediction accuracy, leading to less unpredictable data and thus less access to the unpredictable-data arrays. For the EXAALT data, SZ_prestep runs faster than other solutions because EXAALT data are already aligned and its prediction approach works more effective than SZ_single and ZFP.



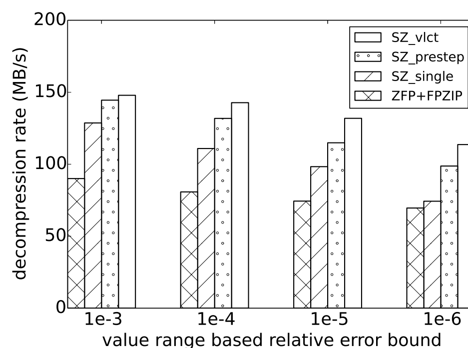
(a) HACC compression rate with fixed value-range-based error bound 1E-4



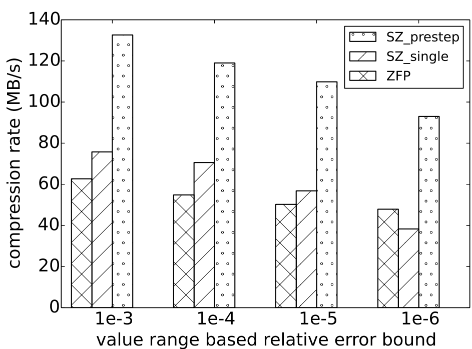
(b) HACC compression rate with fixed pointwise relative error bound 1E-2



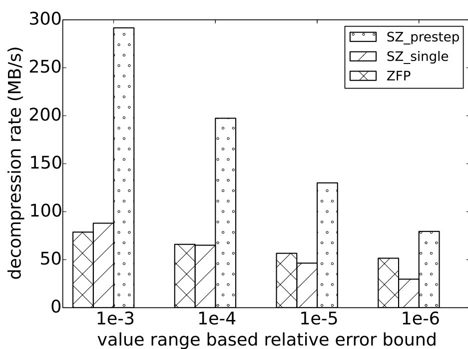
(c) HACC decompression rate with fixed value-range-based error bound 1E-4



(d) HACC decompression rate with fixed point-wise relative error bound 1E-2



(e) EXAALT compression rate



(f) EXAALT decompression rate

Figure 2.9: Compression and decompression rate with different error bounds

Parallel file system performance

In this part, we show that our compression can benefit the overall I/O performance in the parallel environment. Without loss of generality, We took 10 consecutive snapshots out of the 100 HACC snapshots because of the too long original execution time. We performed experiments using 512 cores through 2,048 cores, with up to $36/10 * 2048$ GB = 7.2 TB data loading totally. We compare our compressor to only SZ_single, because other compressors suffer from lower compression ratios. For example, the ZFP+FPZIP takes $56+420+630+64 = 1170$ seconds when running on 512 cores, which is far slower than what we show in Figure 2.10. We adopt the pointwise relative error bound of 0.01 for velocity and the value-range-based error bound of 1E-4 for position, because such settings are satisfying as indicated by the users.

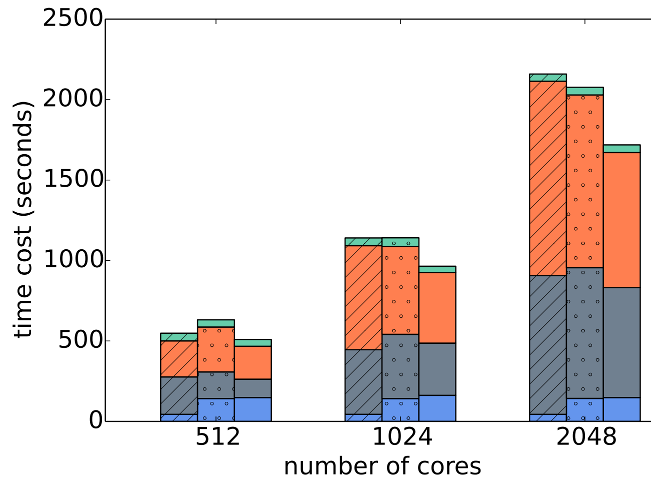


Figure 2.10: Parallel compression/decompression and I/O performance. Slashed is SZ_single; dotted is SZ_prestep; solid is SZ_vlct. The stacked bar represents compressing, writing decompressed data, reading decompressed data, and decompressing time from bottom to top.

As Figure 2.10 shows, although our proposed compressor runs slower than SZ_single, the overall performance gain increases as execution scales. The overall performance with 2,048 cores is improved by 20.39% compared with SZ_single. We also observe that SZ_single outperforms SZ_prestep when running on 512 cores, while they have comparable performance when running on 1,024 cores and SZ_prestep even outperforms SZ_single on 2,048 cores. This means that compression ratio dominates the performance gains than does compression/decompression rate in large-scale I/O-intensive executions. So, the performance gain would increase at larger execution scales.

2.9 Related Work

Many compressors have been developed to save the storage space and I/O cost for today's research based on simulation or instrumental data. The compressors generally can be categorized into two groups: lossless compression and lossy compression. Since scientific data are generated mainly in the form of floating-point values each with random ending mantissa bits, generic lossless binary-stream compressors such as Gzip [5] and BlosC [3] cannot work effectively. Although some existing lossless compressors [20, 77, 92] have been designed for floating-point datasets, they still suffer from limited compression ratios. Hence, the error-controlled lossy compressors [24, 29, 30, 42, 53, 55] have been studied for years, to significantly reduce the data size [90].

Most of the existing lossy compressors [29, 30, 42, 53, 55, 61], however, are designed based on single snapshots, relying on the smoothness of the data in space. SZ [29, 30, 49, 72], for example, predicts the value for each data point based on its neighboring values (in

1D, 2D, or 3D space) and performs a linear-scaling quantization method followed by a customized Huffman encoding algorithm. ZFP [53] splits the whole dataset into many equal-sized small blocks and performs an orthogonal data transform in each block, followed by an embedded coding. Although Isabela [42] does not rely on the smoothness of the data in space because it sorts the data before performing the spline interpolation method, it suffers from a very low compression/decompression rate (generally one or two orders of magnitude slower than other compressors) because of its expensive sorting operation.

The N-body simulation data have very low smoothness in space, as shown in Section 2.4.2 and Section 2.4, leading to low compression quality for the single-snapshot based compressors, as confirmed in our experiments. Although some compressors [24, 41, 86] were designed based on the data smoothness in the time dimension, they are hard to apply in N-body simulations. Numarck [24], for instance, computes the value difference of the data points across two consecutive time steps and explores multiple quantization approaches (such as equal-sized bins, k-means bins, and top-k bins) to maximize the compression ratios. It focuses on the mesh data, which have no alignment issue, unlike N-body simulation data that have different orders and numbers of particles across snapshots. Another compression method [41] designed for MD simulation datasets applies principal component analysis and discrete cosine transform on a relatively large number of consecutive snapshots (e.g., 8∼512 snapshots in their experiments), which is not suitable for cosmological data because of the huge memory cost. In addition, trajectory-based compressors have been developed [57] for MD simulations; however, these compressors are reduced to single-snapshot-based compression when the number of particles varies across snapshots. Moreover, such compressors

require the original data size to be relatively small in order to fit the memory capacity because multiple snapshots have to be collected before the compression [68].

Compared with the existing time-based compressors [24, 41, 87], our compression model is a lightweight, efficient error-bounded compression model for N-body simulation datasets, with limited memory overhead because it relies at most one previous snapshot for the compression of current snapshot. We also design an efficient particle alignment algorithm and improve the data prediction accuracy by leveraging the correlation between the position field and velocity field. Moreover, we optimize the frequency of the space-based compression vs. time-based compression in order to reach a high overall compression ratio, which is a significant step compared with traditional time-based compression approaches such as Numarck [24, 87].

2.10 Summary

In this chapter, we propose a new compression model combining space-based compression and time-based compression as well as a set of optimization techniques. We perform comprehensive evaluations based on two well-known N-body simulation codes (HACC and EXAALT) by comparing our solution with the existing state-of-the-art related work. We summarize the critical evaluation results as follows.

- Our proposed compression method achieves up to 3 times compression ratio improvement on the position field and up to 43% improvement on the velocity field compared with the second-best compressor.

- The overall parallel I/O performance using our compressor outperforms the second-best compressor by up to 20%, by dumping/loading cosmological data on the Argonne Bebop cluster.

Chapter 3

SDC Resilient Error-bounded Lossy Compressor

The previous chapter has shown how incorporating and improving lossy compressor for scientific simulations can speedup the execution. However, the added lossy compressor may experience soft errors which can corrupt the simulation data written to the storage system. Thus, this chapter proposes a soft error resilient version of the lossy compressor, SZ, using ABFT with low performance overhead.

3.1 Introduction

Error-bounded lossy compressors [29, 50, 53, 55, 73] have been effective in significantly reducing large volumes of data produced by scientific simulations [15, 34, 67] or instruments/devices [32, 58], while controlling the data distortion based on user's requirement. Accordingly, error-bounded lossy compression has been thought of as one of the best

ways to resolve today's big science data issue. Silent data corruptions (SDC), however, are nonnegligible when running lossy compressors, as discussed below.

- If one lossy compressor is employed by a high performance computing (HPC) application, it will likely need to deal with a vast volume of data produced by extreme-scale simulations. Various possible failures/errors have to be taken into account. Many existing solutions such as multi-level checkpointing/restart (CR) mechanism focus only on fail-stop issues that are perceived by hardware or operating systems. By contrast, the soft errors, a.k.a. silent data corruption (SDC), may change the data in memory, cache or even register silently, because of inevitably unexpected malfunctions in the system components. Such errors are more dangerous than fail-stop issues, because they may cause biased results in the end of simulation silently.

- 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). However, the devices (such as interplanetary space probe) deployed in space would be more error-prone than the regular devices on the earth. To address this issue, some fault tolerance techniques [37, 52] have been proposed for specific algorithms such as matrix multiplication and FFT. However, when lossy compressors are used by the space systems 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.

There are no lossy compressors designed particularly in the consideration of the possible SDCs. Mat Noor and Vladimirova [59] made a parallel fault-tolerant Integer KLT implementation for lossless hyperspectral image compression on board satellites. Unlike the lossless compression, designing SDC detection/correction method for lossy compression is more challenging since decompressed data will deviate from original data even though there is no SDC.

In our work, we propose the SDC resilient lossy compression based on SZ [50] - one of the best generic error-bounded lossy compressors for large-scale scientific datasets verified by many studies [50, 56]. Not only can our solution detect the SDCs during the compression/decompression but it can also automatically correct the SDCs in some cases.

In general, the SDC errors can be classified into two categories, memory error and computation error, and our solution can protect SZ against both of the two errors. The memory error is introduced by soft errors that corrupt a data value in memory from a to a'. The computation error is introduced by soft errors in logic unit which yields wrong computation results such as $1 + 1 = 3$.

The main idea is analyzing each subroutine in the SZ framework elaborately and designing a series of fault tolerance strategies carefully, such that the lossy compressor can be protected against SDCs 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 SDCs. The SDC errors striking these parts

may cause wrong decompressed data. Thus, it is critical to protect those parts by specific fault tolerance strategies.

- We propose an efficient SDC resilient lossy compression solution based on the SZ compression framework. e reorganize the SZ compression model 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 SDCs on the decompressed data. On the other hand, we design a series of SDC resilient strategies based on SZ's principle, which can not only detect SDCs in most of cases but also correct SDCs in some cases.

- We implement our SDC resilient compressor based on our elaborate design. We evaluate its fault tolerance ability in the presence of SDCs and the corresponding overhead in the fault-free situation, as well as the possible impact to the compression quality. We perform the experiments with real-world simulation data across multiple science domains and image data which were taken by New Horizons space probe [6] in the space. Experiments show that our designed independent-block based compression model has very limited execution overheads (≤10% in most cases). On the other hand, 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 the resilience for SZ (92% running cases with correct decompressed data compare to only 71.2% and 47% of the original SZ).

We organize this chapter as follows. In Section 3.2, we formulate the research problem in terms of the SZ compression framework. In Section 3.3, we provide an in-depth analysis of the fault tolerance ability of SZ. In Section 3.4, we present our fault tolerance methodology. Then we evaluate our methods in Section 3.6 and in Section 3.7, we discuss related work. Lastly we conclude the chapter.

3.2 Background and Problem Formulation

3.2.1 SZ Lossy Compression Framework

SZ [50] is an error-bounded lossy compressor designed for scientific data. According to the recent studies [50, 56, 73], 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 [10]. In the data prediction step, SZ [29, 50, 73] splits the whole dataset into multiple blocks in the size of 6x6x6 and then perform the compression in each block based on two alternative prediction methods - an improved Lorenzo predictor [36] 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.

3.2.2 Algorithm based fault tolerance (ABFT)

ABFT achieves SDC detection and correction by leveraging the characteristics of the algorithms. In high level explanations, ABFT detects SDCs 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 $a[]$ 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 $a[]$ during the period $[t_0, t_1]$. In order to detect the error, we can leverage a checksum (i.e., Equation (3.1)). Specifically, we can calculate the sum of $a[]$ at t_0 and t_1, respectively. Suppose the two calculated sums are denoted by sum_{t0} and sum_{t1}, respectively. If $sum_{t0} \neq sum_{t1}$, we can conclude there must be an SDC error happening to $a[]$ during the period $[t_0, t_1]$.

$$sum = \sum a[i] \tag{3.1}$$

$$isum = \sum i * a[i] \tag{3.2}$$

In order to locate where the SDC error is in the array $a[]$, we can leverage an extra computation - Equation (3.2). Specifically, assuming the value at index j is corrupted during the time period $[t_0, t_1]$, according to $sum_{t_1} - sum_{t_0} = a[j]' - a[j]$ and $isum_{t_1} - isum_{t_0} = j * (a[j]' - a[j])$, one can derive the SDC location index $j = (isum_{t_1} - isum_{t_0})/(sum_{t_1} - sum_{t_0})$. This ex-

ample illustrates that it is viable to detect and even correct the single-data-point error just by introducing a few more light-weight computations.

3.2.3 Error model and assumptions

We identify the error model in this subsection. In our study, we focus on both 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 [66]. Moreover, it is too difficult to comprehensively detect the FCEs even for professional FCE detection tools according to recent studies [66]. 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 $10\times10\times10$ in size). Similar to other ABFTs [23, 48, 82], we assume the checksum itself is error free because of its tiny computation time compared with the compression time.

3.2.4 Formulation of SDC 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 SDC 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 N) that take the majority of memory footprint in SZ because they are the major objects affected by SDCs 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 chapter.

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 SDC resilient lossy compressor, as listed below.

- *SDC detection/correction ability.* What kinds of SDCs could be detected or corrected? What is the accuracy and coverage rate of SDC error detection?
- *Computational Overhead.* It is defined as the ratio of the extra computation time to the total original execution time in an error-free situation.
- *Impact to Compression Result.* Whether the SDC 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 SDC 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.

3.3 Resilience Analysis of SZ 2.1

In this section, we analyze the resilience (SDC detection/correction ability and impact) of SZ 2.1 based on its principle.

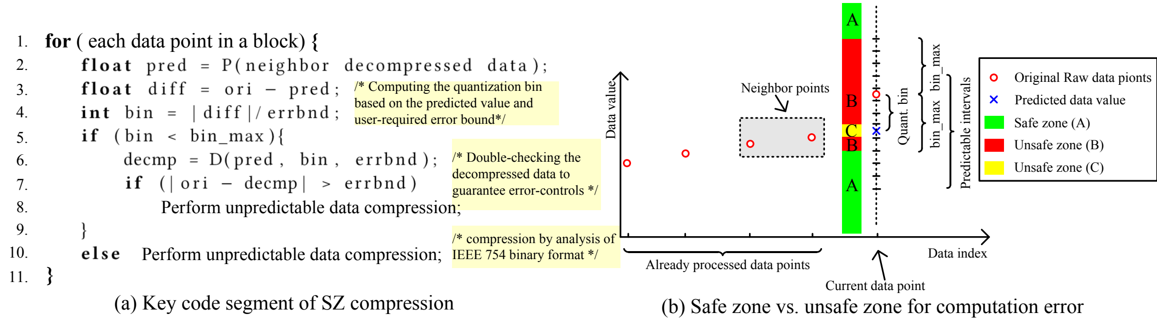


Figure 3.1: Analysis of fault tolerance ability for SZ with computation error

3.3.1 SDC Resiliency – Computation error

We analyze SZ's natural resilience based on when/where the computation error could happen, including calculation of regression coefficients, 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'.

SDC 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 coefficients or incorrect bestfit predictor selection, exactly the same coefficients/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 coefficients or incorrect predictor selection.

SDC 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 Figure 3.1 (a), which is conducted in a loop of scanning all data blocks. It 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 7-8): 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 Figure 3.1 (a), based on five possible cases. We note that 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.

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 3.1 (b)), the decompressed data will still respect the error bound because of the type-2 behavior.

- *Situation 2:* the impact of the SDC 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 3.1 (b)) 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 SDC 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). What is even worse is that this decompressed value would also be used to predict other data points in the decompression, such 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 SDC 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 in this situation 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.

SDC resilience in lossless compression

We will show our solutions are able to detect SDCs that occur in lossless compression in Section 3.4.3.

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 3.1 (a)) and the correctness of data decompression during the compression (i.e., line 6). To address this issue, we developed an efficient selective instruction duplication method, to be described in Section 3.4 in detail.

3.3.2 SDC Resilience – Memory error

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

SDC 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 SDC errors. The input data is used in the following steps: 1. computing the regression coefficients; 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 3.3.1, 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 SDC 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).

SDC resilience against the memory error in regression coefficients

The memory usage of regression coefficients 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 coefficients (for 3D dataset). Thus, the coefficients only take $\frac{4}{blocksize}$ of the overall memory. For a 3D example, usually the block size is 8X8X8 which means the coefficients take only $\frac{1}{128}$ of overall memory.

SDC 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 on (e.g., during the Huffman encoding stage).

3.4 Error Tolerance Methodology

Our SDC resilient SZ 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.

3.4.1 Blockwise independent design

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

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. Both recording the bin array and Huffman encoding need to be done individually per block.

Another significant advantage in the independent-block based compression design is that one can perform random-access decompression efficiently 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 3.6. Moreover, the independent-block based compression also makes the parallelism of SZ much easier to port on many-core architectures, such as GPU [76].

3.4.2 Fault tolerant compression

We present our SDC resilient compression method in Algorithm 3. 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 SDC errors striking the input data later on. As we discussed in Section 3.3.2, we do not need to detect memory error in the input data during computations for regression coefficients 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 sum_{in}), it can be located and recovered by the pair of checksums (i.e., sum_{in} and $isum_{in}$) 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 SDC errors occurring in the decompression stage, to be detailed later. For the computation errors, instruction duplication can be used. According to our analysis in Section 3.3.1, only data prediction (line 18) and calculating decompressed data (line 25) need to be protected by instruction duplication.

Algorithm 3 SOFT ERROR RESILIENT SZ COMPRESSION

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

```
1: for each block (block i) of the input data do
2:     Compute the regression coefficients
3:     Get sum_in[i] on input by Equation (3.1) /*for SDC in input data*/
4:     Get isum_in[i] on input by Equation (3.2) /*for SDC in input data*/
5: end for
6: for each block (block i) of input data do
7:     Sample and estimate E_reg and E_lor
8:     indicator[i] ← the one with smaller error /*regression or lorenzo*/
9: end for
10: for each block (block i) of input data do
11:     Do memory error detection and correction using sum_in and isum_in
12:     if indicator[i] == regression then
13:         f() ← regression predictor
14:     else
15:         f() ← lorenzo predictor
16:     end if
17:     for each data point, ori, in the data block do
18:         pred' ← f_dup() /*f_dup(): instruction duplicated f()*/
19:         diff ← ori − pred'
20:         q_bin ← quant(diff,e) /*get quantiz. bin based on diff,e*/
21:         if q_bin is not in the acceptable bin range then
22:             Compress ori as unpredictable
23:         else
24:             Calculate sum_q, isum_q for q_bin[]
25:             dcmp ← dec_dup(q_bin, pred') /*dec_dup() is instruction duplication based version of dec()*/
26:             if |ori − dcmp| > e then
27:                 Compress ori as unpredictable
28:             end if
29:             sum_dc[i]+=dcmp /*cksum for decompressed data of block i */
30:         end if
31:     end for
32: end for
33: Construct Huffman tree
34: for each block of q_bin[] do
35:     Do memory error detection and correction using sum_q and isum_q
36:     Encode q_bin[] by Huffman tree
37: end for
38: Compress encoded q_bin by lossless method (Zstd)
39: Write compressed q_bin and unpredictable data to byte file
40: Compress sum_dc[] by lossless method (Zstd) and write to file
```

3.4.3 Fault tolerant decompression

The SDC resilient SZ decompression is presented in Algorithm 4. 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., $sum_{dc}[]$) by lossless compression (Zstd) during the data compression. Accordingly, we need to decompress sum_{dc} (line 10) before the error detection. Our idea is leveraging such checksums of decompressed data (i.e., $sum_{dc}[]$) constructed during the compression to detect possible errors that happen during the 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 $sum_{dc}[]$ (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 can conclude that the SDC error likely happens during the lossless compression, which will be reported to users (line 19).

3.4.4 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

Algorithm 4 SOFT ERROR RESILIENT SZ DECOMPRESSION

Input: The SZ compressed file in byte (cmp_data) and compressed *sum* for blocked decompressed data ($sum_{dc}[]$).
Output: Decompressed data with bounded error compared to original data.

1: Decompress cmp_data by lossless compressor (Zstd)
2: **for** each block **do**
3:     $q_bin[] \leftarrow$ decode using Huffman tree
4:     **if** it was compressed by Lorenzo **then**
5:         $dec_data[] \leftarrow$ Lorenzo decompression
6:     **else**
7:         $dec_data[] \leftarrow$ regression decompression
8:     **end if**
9: **end for**
10: Decompress $sum_{dc}[]$ by lossless compressor (Zstd)
11: **for** each block of decompressed data (block index $= i$) **do**
12:     Calculate checksum (denoted sum_i) for this block of $dec_data[]$
13:     **if** $sum_i \neq sum_{dc}[i]$ **then**
14:         Reexecute line 4-9 for this block /*random-access decompression*/
15:         Calculate checksum (denoted sum_i) for this block of $dec_data[]$
16:         **if** $sum_i = sum_{dc}[i]$ **then**
17:             Report: memory/computation error detected but corrected
18:         **else**
19:             Report: SDC in compression; Return
20:         **end if**
21:     **end if**
22: **end for**

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 approximate 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 $10 \times 10 \times 10$ block) in general. With all these techniques, we can provide bit-level error detection and correction. The main difference from Demmel's work [26] 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.

3.4.5 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 R_0; the number of data blocks by n. 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 $CR_decrease = (\frac{R_0-1}{R_0+n-1})\times 100\%$. 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 6X6X6 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.

3.5 Discussion for SZ Time Based Compression

The above resilience analysis only applies to space based compression of SZ. To build an error resilient time based compression of SZ, we need to protect the particle alignment operations which we introduced in Section 2.6 since particle alignment is one of the

dominant part in time based compression. Noticing sorting is the most time consuming operations of particle alignment algorithm, we reduce the resilience problem to be providing an error resilient sorting algorithm. Though the radix sort we used in Algorithm 1 has no efficient soft error resilience, we can use introsort since it has much better error resilience and uses much less memory space. We present how to achieve efficient fault tolerance for introsort as follows.

3.5.1 Introsort

Introsort is a hybrid sorting algorithm built upon quicksort and heapsort; introsort achieves the best average performance of quicksort ($O(n \log n)$) but avoids the poor performance of quicksort in the worst case ($O(n^2)$) by switching to heapsort (a guaranteed $O(n \log n)$). We review these two building blocks before presenting the introsort algorithm.

Quicksort. We use the basic quicksort algorithm presented by Lomuto [17] in our discussion for simplicity. It should be noted that our analysis is valid for quicksort variations [11, 18] as well; improvements to aspects such as pivot selection do not impact our analysis. Quicksort is a divide and conquer algorithm that sorts a given n-element array, $a[]$. Quicksort chooses the first element of the given array as the *pivot*, around which the array is partitioned.

Quicksort chooses $a[0]$ as the pivot to partition the original array into two sub-arrays, $[\, a[1],\ a[2],\ a[3],\ ...,\ a[j]\,]$, with each element strictly less than $a[0]$, and $[\, a[j+1],\ a[j+2],\ a[j+3],\ ...,\ a[n-1]\,]$, with each element no less than $a[0]$. The pivot $a[0]$ is then swapped to a medial position between the two sub-arrays. These two steps are then recursively applied to the two sub-arrays, until each sub-array contains at most one element.

After each partitioning, the pivot rests in what will be its final position in the sorted array, and thus it remains untouched in later recursive sorting.

Heapsort. Heapsort is built on the idea of a *max-heap*: a complete binary tree where the value of each non-leaf node is greater than or equal to the value of any first-generation child. Heapsort represents a max-heap as an array, with the root as the first element of the array; the first element is therefore maximal.

Heapsort consists of two stages: max-heap creation and sorting using the max-heap. In the first stage, the given n-element array $a[]$ is adjusted to form a max-heap. The second stage is an iterative process of excluding the largest element and adjusting the max-heap for the remaining elements. In the first iteration, the largest value, $a[0]$, is excluded by swapping it with $a[n-1]$, and then adjusting the first $n-1$ elements to form a new max-heap; the largest element is then found at the end of the array. The second iteration repeats the process on the (n-1)-element max-heap formed from the remaining (n-1) elements of the array, swapping out the second largest value $a[0]$ with $a[n-2]$ and adjusting a new (n-2)-element max-heap, and so on for the remaining iterations. The array is sorted in ascending order at the end of this process.

Introsort. The GNU Standard C++ Library is internationally recognized, and introsort is the GNU C++ default sorting routine. We base our discussion and implementation of introsort on the version found in this library. Our fault tolerant introsort is thus readily available to a large user base.

GNU C++ users can invoke introsort by calling `std::sort`. Algorithm 5 shows the code skeleton of `std::sort` and the first major phase, `introsort_loop()`. While in

Algorithm 5 *std::sort*(a[], n)

1: **if** $n \leq 1$ **then return**
2: **end if**
3: introsort_loop(a[], $2 \log n$)
4: insertion_sort(a[], n)
5: **return**

Function: introsort_loop(a[], depth_limit):
while a[].current_partition.size $>$ 16 **do**
  if depth_limit = 0 **then**
    heap_sort(a[].current_partition)
    return
  end if
  depth_limit ← depth_limit - 1
  P ← pivot(a[].current_partition)
  [L[], R[]] ← partition(a[].current_partition, P)
  introsort_loop(R[], depth_limit)
  a[].current_partition ← L[]
end while

`introsort_loop()`, quicksort is used to recursively partition the given array either until all partitions are 16 elements or less, or until the recursion depth is deeper than the pre-defined depth limit of $2 \log n$, where n is the size of the array. When the recursive partitioning stops due to hitting the depth limit, `introsort_loop()` uses heapsort to sort the remaining partitions with more than 16 elements, avoiding the $O(n^2)$ worst-case complexity of quicksort.

Once the first phase completes, all partitions with 16 elements or less remain unsorted. These small, unsorted partitions are called *leaf partitions*. After `introsort_loop()` completes, the second major phase, `insertion_sort()`, uses an insertion sort on the entire array; the leaf partitions are of an optimal size for insertion sort to perform well, and a completely-sorted array is returned at the end.

Figure 3.2 illustrates both phases of the introsort sorting process with a recursion depth limit of 4.

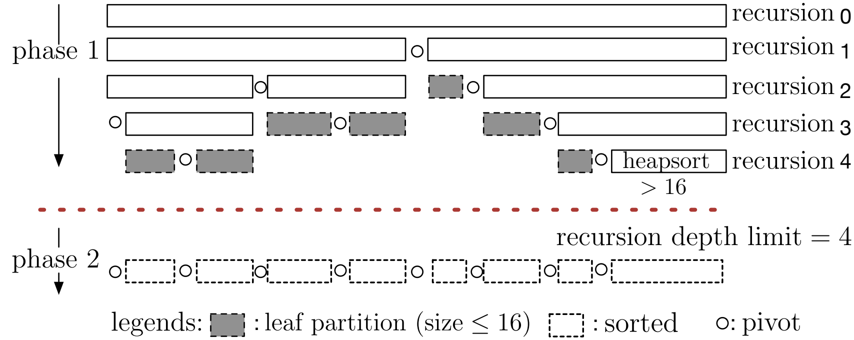


Figure 3.2: Example introsort execution; phase 1 comprises quicksort and heapsort; phase 2 executes insertion sort.

3.5.2 Comparison Errors

Paired comparison is the operation of comparing two values, resulting in a true or false `bool` value output; it is the most common basic computation operation of introsort. A *comparison error* occurs when soft errors corrupt the result of a comparison instruction either by altering the instruction operation code (opcode) or the register operands [8]. For example, a comparison instruction $a < b$ may return an incorrect result if the operation is changed to another operation, such as $a + b$; it may also return an incorrect result if the register file storing the real result of $a < b$ is corrupted.

Most algorithm-based fault tolerance (ABFT) studies target computation errors only related to the *key data structures* of the algorithm under consideration [82,83], ignoring the computation errors involving control-flow data. For example, the ABFT strategy for matrix multiplication algorithms focuses on the computations errors in the matrix multiplication only, without considering computation errors in the calculation of iteration counts. For our work, we likewise focus on computation errors (comparison errors in the context of sorting) arising from the comparisons of elements in the given *array*.

3.5.3 Efficient Error Resilience for Introsort

E-sorted-merge. We name the following procedure as *e-sorted-merge*.

1. Scan the output and record inversion pair locations;

2. If no such location is found, return; else go to step 3;

3. Merge the sorted segments partitioned by the above locations recursively until a single sorted array is generated.

The efficient error resilient introsort is outlined as follows:

- Apply TMR-protected *e-sorted-merge* to the output of heapsort in phase 1.

- Apply TMR (Triple Modular Redundancy) to insertion sort in phase 2.

All errors occurring within quicksort in phase 1 are left unattended and are corrected in phase 2 by a TMR-protected insertion sort. This strategy avoids the unnecessary overhead of TMR-protection for quicksort and heapsort. This algorithm is proved to have very low overhead in both theoretical analysis and empirical analysis. The detail can be found in the paper [47].

3.6 Experimental Evaluation

3.6.1 Experimental Setup

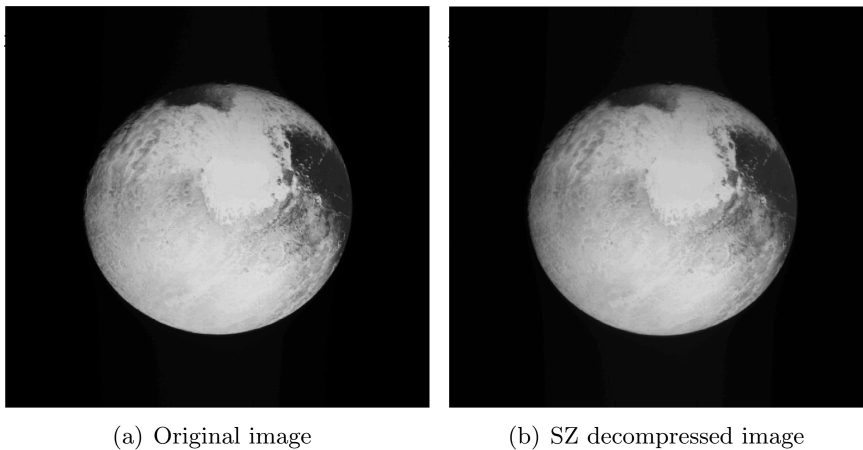
In this subsection, we describe how we set the experiments in our evaluation, including applications, error injections, and experimental environment.

Table 3.1: Basic dataset information

Dataset	# Fields	Dimensions	Science
NYX	6	512X512X512	Cosmology
Hurricane	13	100X500X500	Climate
SCALE-LETKF (SL)	6	98X1200X1200	Weather
NASA: Pluto	1	1028X1024	Aerospace

Applications

We evaluate our SDC 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) [7]. Those images were taken by New Horizons space probe [6] in aerospace which is an error-prone environment because of potential impact of cosmic rays. The description to these datasets is presented in Table 3.1. 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 3.3.



(a) Original image (b) SZ decompressed image

Figure 3.3: Visualization of Original Data vs. Decompressed Data (Pluto photo taken by New Horizons [6]; SZ compression using Value-range based error bound: 1E-3)

Error injections with two modes

Evaluation mode A - source-code level error injection. Like most ABFT work [23, 48], 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 calculated on input data. To simulate the computation errors when calculating regression coefficients, sampling and estimating compression error of Lorenzo and regression, we randomly select a data point in a random block and 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) [12] 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) [2], 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 the process by the bit-flipped checkpoint. We inject 1, 2 or 3 errors and perform 500 runs per test for both fault tolerant SZ and original unprotected SZ.

Experimental Environment

We run experiments on a supercomputer [1]. Inside each computing node are two Intel Xeon E5-2695 v4 processors totalling 36 cores. The POSIX I/O [81] 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.

3.6.2 Evaluation of Independent-block Compression

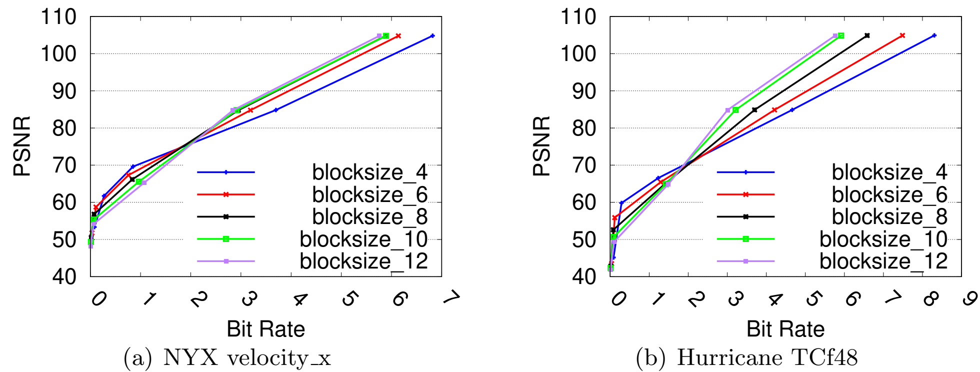
We first evaluate our designed independent-block based SZ compression (a.k.a., random-access based compression).

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 3.4. 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 $\leq$2); large block sizes (such as 8x8x8 $\sim$

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-coefficients 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 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 relatively low bit-rates.



(a) NYX velocity_x (b) Hurricane TCf48

Figure 3.4: Rate distortion with different block sizes

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 3.4.3, this design can also help correct the errors very quickly upon a detection of problematic blocks by checksums. In Figure 3.5, we present the decompression

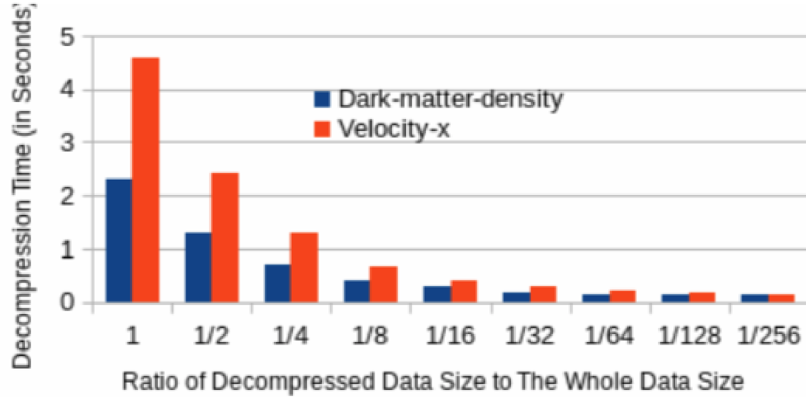


Figure 3.5: Efficiency of random access decompression

Table 3.2: 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%

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 efficiency of random-access decompression.

3.6.3 Error free experimental results

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

Table 3.3: Percentage of runs whose maximum absolute error is within error bounds in sz and ftrsz

	injecting errors in input data			
	Successful runs with correct decompressed data			
error bounds:	1E-3	1E-4	1E-5	1E-6
sz	60%	57%	49%	48%
ftrsz	100%	100%	100%	100%

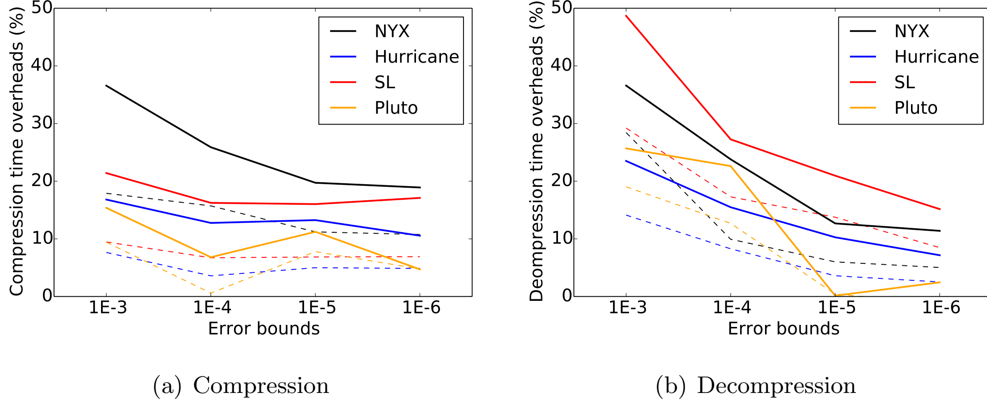
	injecting errors in quantization bin array							
	Successful runs with correct decompressed data				Normal runs without segmentation faults			
error bounds:	1E-3	1E-4	1E-5	1E-6	1E-3	1E-4	1E-5	1E-6
sz	3%	1%	1%	0%	34%	34%	49%	54%
ftrsz	100%	100%	100%	100%	100%	100%	100%	100%

Compression ratio overhead

Since we store the checksum $sum_{dc}[]$ during the compression in order to verify the correctness of the decompressed data, the compression ratio could be degraded more or less. Table 3.2 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.

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



(a) Compression

(b) Decompression

Figure 3.6: Compression time and decompression time overheads. Dash lines are random access SZ; solid lines are fault tolerant random access SZ.

in Figure 3.6. We can see from Figure 3.6 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.

3.6.4 Error injected experimental results

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 ensure the decompressed data within the user defined 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.

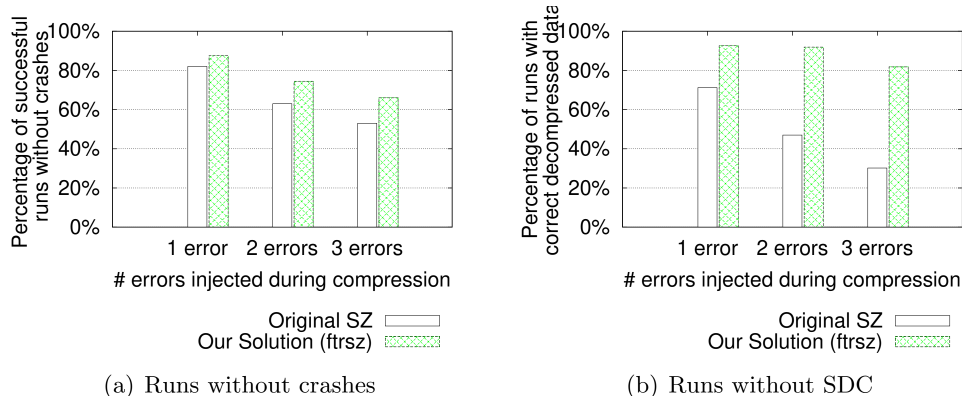
As shown in Table 3.3, our proposed fault tolerance solution can always yield correct decompressed results when the memory errors are injected in input data or quantization bin array. The 100% correctness of the decompressed data under ftrsz also means that our solution is immune to the round-off errors. In comparison, for the original SZ without our techniques, 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 lower part of Table 3.3, 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.

Resilience against memory errors happening anywhere (evaluation mode B)

Figure 3.7 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 3.3. In addition, as shown in Figure 3.7 (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 injection 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) Runs without SDC

Figure 3.7: Experimental results using evaluation mode B

Resilience against computation errors during compression

As discussed in Section 3.3.1, the computations of regression coefficients, sampling and estimating compression error are error resilient though computation errors will impact the compression ratio. Figure 3.8 shows our experimental results about the impact to compression ratios. Computation errors are randomly injected and each experiment is repeated

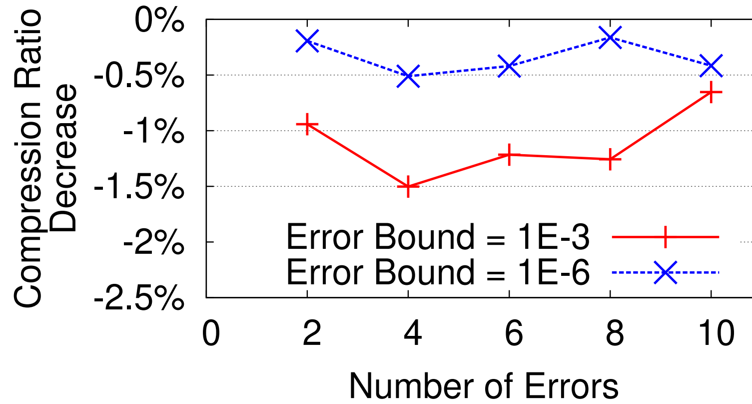
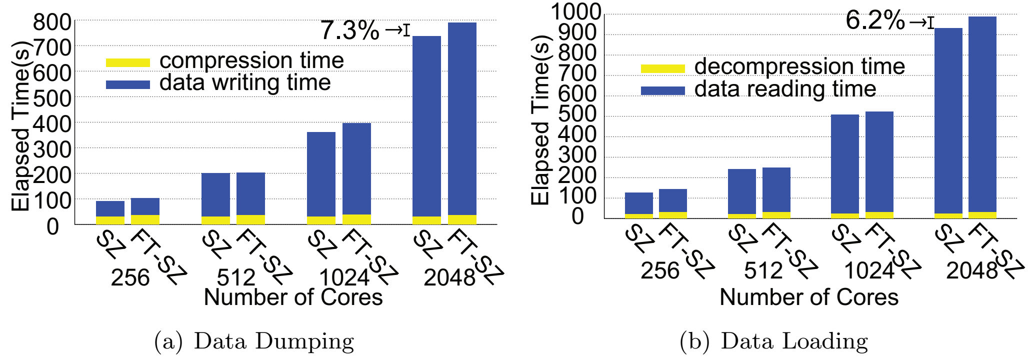


Figure 3.8: Compression ratio decrease with cmput. errors

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 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.

Resilience against errors injected during decompression

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.



(a) Data Dumping

(b) Data Loading

Figure 3.9: Performance of data dumping/loading (sz vs. ftrsz)

3.6.5 Parallel experimental results

We evaluate the I/O performance with breakdown of the execution times (compression/decompression time + data writing/reading time) by processing NYX dataset under the error bound of 1E-4 in parallel on the PFS of the cluster. The experiment follows a weak-scaling style: i.e., we run the tests 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 3.9. 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 2,048 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.

3.7 Related Work

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

So far, there have been many lossy compressors [24,29,42,50,53,55,67,73–75] developed to significantly reduce the large volume of data or checkpoint file produced by scientific simulations. All the lossy compressors, basically, could be classified into two categories - transform-based compression [53,67] and prediction-based compression [29,50,55,73]. None of the transform-based compressors are immune to the SDCs. 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 SDC issue could be also fatal to the reconstruction of data. In SZ, for example, if the data prediction on some data point is corrupted 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 the memory error and computation error, respectively. From the perspective of hardware, error correcting code (ECC) has been implemented to detect and correct 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 SDCs has motivated algorithm based fault tolerance (ABFT) [35], 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 [35]. 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.

3.8 Summary

In this chapter, we propose a novel SDC resilient strategy for the SZ lossy compressor. We develop an independent-block based compression model for SZ to improve the robustness. We analyze each subroutine of the SZ framework elaborately 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. Our solution can control the time overhead to about 10%, with a degradation of compression ratio limited within about 5%. When injecting one and two SDC 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).

Chapter 4

Towards End-to-end SDC Detection for HPC Applications Equipped with Lossy Compression

In this chapter, we present our application-scope soft error detection schemes for scientific simulations equipped with lossy compression. Different from the previous algorithm-scope scheme for lossy compression in chapter 3, application-scope scheme covers all soft errors happening from the start of the scientific simulation to the end. Thus, it is also called end-to-end detection.

4.1 Introduction

Science data compression techniques have been widely demanded and used by today's large-scale scientific HPC applications, as these applications are producing extremely

large volume of data. Lossless compressors are not suitable for compressing science data in that the science data are mainly composed of floating-point values which involve disordered ending mantissa bits in their binary representations. Error-bounded lossy compression has been studied for years, since not only can it significantly reduce the data size but it can also strictly control the data distortion based on user-specified error bound. Nowadays, error-bounded lossy compressors have been broadly verified as very helpful to saving storage space and improving I/O performance for many production-level applications across different science domains, such as cosmology simulations [34, 60], MD simulations [4], climate simulations [25, 39], and quantum computing simulations [38].

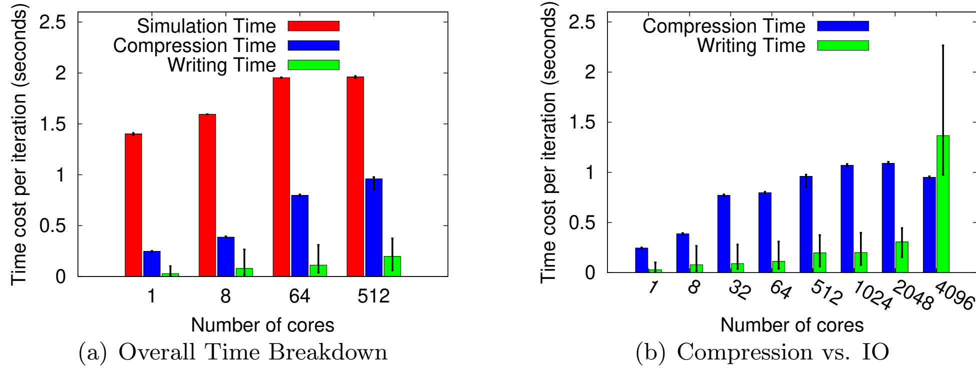
Since the large-scale HPC applications equipped with the error-bounded lossy compressors need to deal with vast amount of data through various devices (CPU, cache, memory, NIC, I/O and storage), silent data corruptions (SDC) are unavoidable. According to the cosmologists, HACC cosmology simulations [34] may produce dozens of petabytes of data when simulating 1 trillion particles for hundreds of time steps (or snapshots). Quantum computing simulation [85] may produce up to 32 exabytes of data which need to be compressed and decompressed during the simulation because of inadequate memory space (e.g., today's fastest supercomputer - ORNL Summit [85] - has only 2.8 PB of memory capacity in total). In fact, SDCs would happen more frequently if aggressive power saving techniques are used [71, 83, 84].

Generic Data-analytic Based Fault Tolerance methods (DBFT) [16, 27, 28] have been exploited for years to protect the large-scale HPC applications against SDCs. Compared with DBFT, other SDC detection solutions suffer from many constraints, significantly

limiting their usability in practice. Algorithm-Based Fault Tolerance (ABFT) [35], for example, have been extensively studied for adding fault tolerance in multiprocessor architectures. However, each ABFT approach is relying on the inner mechanism of some numerical algorithm such that it can only be used on some specific algorithm. Replica Based Fault Tolerance (RBFT) is another SDC detection solution, which creates the replica processes to detect/correct possible errors. RedMPI [31] is a typical example; it leverages "replica" MPI tasks and performs online message verification intrinsic to existing MPI communication. However, RBFT suffers from significant redundancy of resources ($2\times$ for detection and $3\times$ for correction). Many recent studies [22, 28] have demonstrated that DBFT can effectively detect the SDCs for HPC simulations with very limited time overhead and no resource redundancy, making it a very appealing solution to users in practice.

None of the existing DBFTs support/consider the HPC applications equipped with error-bounded lossy compressors, while the data processing time (including data compression and writing) is non-negligible compared with the simulation time. Figure 4.1 presents the time breakdown per time step when running the FLASH-Sedov simulation [21] on Bebop [1] in terms of weak-scaling (problem size $64 \times 64 \times 64$ per core) with the compression ratio of about 7:1 under SZ. It is observed that the simulation time increases little with the exponentially increasing problem sizes (from totally $64\times64\times64$ to $512\times512\times512$), meaning a fairly good scalability. By comparison, the compression time also increases little, in that each process is compressing the fixed amount of data ($64 \times 64 \times 64$). We can clearly observe that the data writing time increases significantly with larger problem sizes (especially when the execution scale is larger than 4096 cores) because of the limited number of I/O nodes

on Bebop. In this experiment, each case is conducted 10 times, and we observe that the I/O time is fairly unstable, which is because all the jobs are sharing the I/O nodes for data reading/writing. In fact, the compression speed could be fairly slow especially when the error bound is relatively small (as low as only 10MB/s, as demonstrated in [65]). Our prior work [51] also showed that the compression time could be about 5× slower than the data writing time on the ANL Theta [70] - a supercomputer ranking as 34th in the recent top500 list. All in all, protecting the data processing phase (including both compression and writing) is critical to guaranteeing the correctness of the snapshot results outputted in the course of simulation.



(a) Overall Time Breakdown

(b) Compression vs. IO

Figure 4.1: Analysis of time cost in the parallel simulation - FLASH (Sedov)

Developing a lossy-compression supported DBFT is a new yet non-trivial research problem. First, many different DBFT approaches [16, 27, 28] have been proposed and each has its own pros and cons, with respect to detection ability, execution overhead, memory cost, etc. Second, HPC applications may store the big data in different ways, so that the memory layout and storage performance could differ a lot with different applications. For example, the applications may write the data either synchronously or asynchronously.

Under the synchronous writing mode, the original data would be kept unchanged in memory during the whole data writing procedure, such that the regular DBFT can be applied on the original data. However, under the asynchronous writing mode, the data could be refreshed by the simulation immediately as the data writing gets started. In this case, DBFT can only be applied on the lossy compressed data because of missing original simulation data. Third, the science data could be compressed by different error-bounded lossy compressors [29, 53, 73], which may have different characteristics on compression errors [15, 54]. This brings a grand challenge to detect SDCs occurring during the compression procedure.

In this work, we exploit efficient SDC detection methods to protect the entire duration related to simulation data (from data producing through data saving at each time step) for HPC applications equipped with lossy compression. The contribution is below.

- We develop a series of DBFT-based end-to-end SDC detection methods for HPC applications equipped with error-bounded lossy compressors. Our study is based on the state-of-the-art generic SDC detector - Adaptive Impact-driven Detector (AID) [28] because it exhibits an excellent detection ability and very limited time/resource overhead in the literature [40, 69, 79].

- We analyze the pros and cons for our proposed DBFT-based end-to-end SDC detectors. We also identify the most appropriate solution in different situations with various I/O modes, different data dumping periods, and I/O costs.

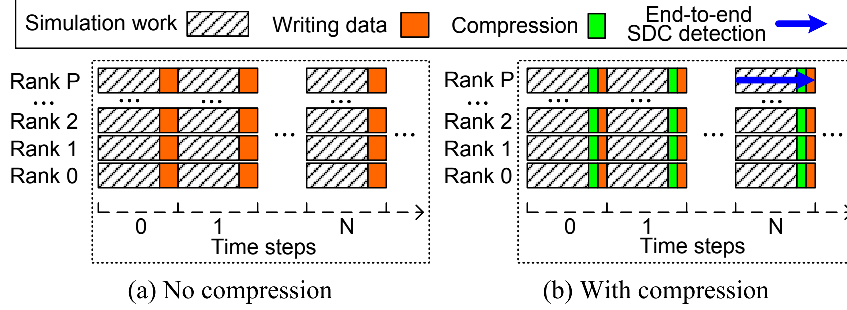
- We thoroughly assess our proposed SDC detection methods using four well-known scientific simulations using two state-of-the-art error-bounded lossy compressors running on Argonne Bebop supercomputer [1] with up to 1,024 cores.

The remainder of the chapter is organized as follows. Section 4.2 formulates the research problem. Section 4.3 discusses the preliminary concept and background. Section 4.4 describes the end-to-end SDC detection methods which combine different DBFT algorithms and different error-bounded lossy compressors in different I/O modes (synchronous vs. asynchronous). Section 4.5 assesses SDC detection methods (including performance overhead, detection ability, etc.) using four production-level HPC simulations and datasets. Section 4.6 discusses the related work. We conclude the chapter in Section 4.7.

4.2 Problem formulation

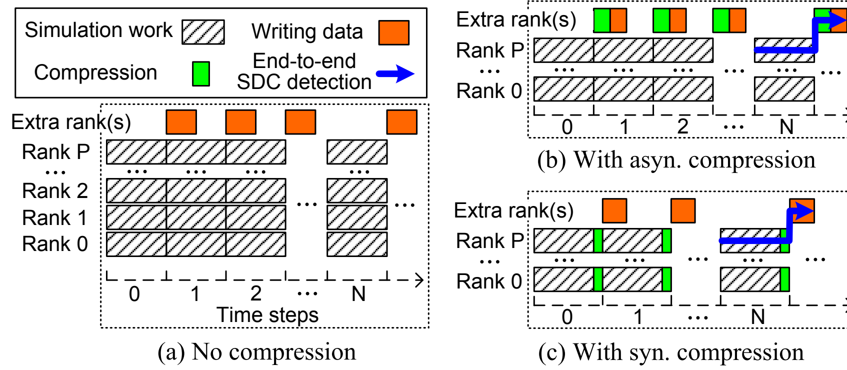
In this section, we formulate the research problem. Given a large scale scientific simulation with multiple time steps (or iterations) equipped with error-bounded lossy compressors, our target is to protect its execution against any possible SDCs happening in any part throughout its whole lifetime, including simulation, data compression and data writing.

Basically, there are two execution modes based on either synchronous or asynchronous I/O adopted by parallel applications, as illustrated in Figure 4.2 (a) and Figure 4.3 (a). The key difference between these two modes, with respect to the SDC detector, is that the former generally keeps the variables' data unchanged in memory throughout each whole time step; while the latter would refresh the variables' data immediately after they are transferred to the extra rank(s) for asynchronous data writing. This would lead to a significant change to the data-analytic based SDC detection method, especially when considering the error-bounded lossy compressors.



(a) No compression (b) With compression

Figure 4.2: Parallel simulation with synchronous I/O



(a) No compression (b) With asyn. compression (c) With syn. compression

Figure 4.3: Parallel simulation with asynchronous I/O

As mentioned previously, the HPC applications may adopt either synchronous or asynchronous I/O to perform data writing. Similarly, the data compression can also be performed either synchronously or asynchronously, depending on user's demand, compression time, simulation time, and/or I/O bandwidth. Figure 4.2 (b), Figure 4.3 (b) and Figure 4.3 (c) illustrate the three different situations.

Our research goal is to explore the bestfit end-to-end SDC detection method for the HPC applications equipped with lossy compressors, in terms of the outstanding SDC detector - AID [28]. The end-to-end SDC detection means that the SDC detection method should protect the execution throughout the whole time period including simulation, data compression and data writing for each time step. We illustrate the end-to-end detection for

one running process/rank in three different cases (please see blue arrows in Figure 4.2 (b), Figure 4.3 (b) and (c)).

4.3 Preliminary Concept and Background

In this section, we describe the preliminary concept and background which are fundamental to the design of end-to-end SDC detection methods. As mentioned in the problem formulation, the end-to-end SDC detection would involve two key libraries, adaptive impact driven SDC detector (AID) and error-bounded lossy compressors, to be discussed as follows.

4.3.1 Adaptive Impact Driven SDC Detector (AID)

Adaptive impact driven SDC detector (AID) [28] is a generic data-analytic based SDC detector, whose high detection ability has been extensively verified in literature [22,79]. There are two critical concepts in the AID as listed below.

- **Adaptive detection**: AID is an adaptive detector. It adopts one-step ahead prediction to predict the value for each data point of the key variables at each time step and compare the observed values with confidence intervals located by the predicted values. The critical design in AID is that it dynamically selects the bestfit feedback-control prediction methods for different ranks based on their local runtime data. This can significantly improve the detection ability while reducing the memory overhead to $2\times$ in general cases.
- **Impact-driven detection**: AID is an impact-driven detection method, because it aims to detect only influential SDCs in terms of dynamic HPC data features. Specif-

ically, AID allows users to set an impact error bound ratio (denoted by θ) for each application run. The influential SDC at time step i is defined as the change of the data whose errors exceed the threshold $\theta \cdot range_i$, where $range_i$ refers to the data value range of time step i during the simulation. We carefully characterized the impact of SDCs (i.e., impact error bound ratio) based on 18 real-world applications across from different domains such as burn simulation, N-body simulation, hydrodynamics and heat diffusion [28].

Note that although AID focuses only on the smoothness of the data stored in the memory, it can cover both memory errors and computation errors in principle, because of the impact-driven design. In fact, the major part of the memory is composed of the state variable data (such as density, pressure, and temperature) used for advancing the simulation in each iteration (or time step). Those state variable data also constitutes the major portion of the checkpointing files used to restart the applications upon failures/interruptions [67]. No matter what kinds of computation errors happen during the simulation, the subsequent simulation must be correct as long as the state variable memory is not affected according to the impact error bound ratio. Otherwise, the application would be restarted from the latest SDC-free checkpointing files to correct the suspicious SDCs.

4.3.2 Error-bounded Lossy Compression

Error-bounded lossy compression allows users to strictly control the data distortion for the lossy compression based on a user-specified error bound. Given a dataset (denoted by $D = \{d_1, d_2, \cdots, d_N\}$) with N floating-point data values, absolute error-bounded com-

pression requires the reconstructed data (denoted by $D' = \{d'_1, d'_2, \cdots, d'_N\}$) must respect a constant bound (denoted by e), as shown in Formula (4.1).

$$|d_i - d'_i| \leq e, \forall d_i \in D, d'_i \in D' \tag{4.1}$$

Error-bounded compression is widely used by domain researchers to compress big volume of data based on the error bound generated by an offline analysis of the relationship between the error bounds and target post-analysis metrics [14,15]. With the error bounded compressor, the simulation data volume can be significantly reduced, while the data fidelity can still be guaranteed. The typical state-of-the-art error-bounded compressors include SZ [29,50,73,91] and ZFP [53].

4.4 Data-analytic based End-to-end SDC Detection

In this section, we present various design strategies for end-to-end SDC detection in the scientific simulations equipped with lossy compression techniques. Before introducing the end-to-end detection methods, we need to describe some key definitions and notations, as follows.

Table 4.1 lists the key notations to be used in the following text. As mentioned previously, the original AID algorithm needs to predict each data point for the current time step i based on the corresponding values at last 3 time steps ($i-1$, $i-2$, and/or $i-3$). We use S_i to denote the current snapshot, and use S_{i-1}, S_{i-2}, and S_{i-3} to denote the previous snapshots at time steps $i-1$, $i-2$, and $i-3$, respectively. Similarly, D_i refers to the decompressed data of time step i, and D_{i-1}, D_{i-2}, and D_{i-3} refer to the decompressed data

in the previous time steps. For simplicity of description, we use P_i (=$\{p_{i1}, p_{i2}, \cdots, p_{iN}\}$) to denote the predicted data at time step i, which will be used to determine whether the simulation data have suspicious SDC errors. Obviously, the higher the prediction accuracy for each data point, the higher detection ability or accuracy the SDC detector would have.

Table 4.1: Table of Key Notations

Notation	**Description**
S_i	simulation snapshot to be protected at current time step i
S_{i-1}	original simulation snapshot at time step $i-1$
S_{i-2}	original simulation snapshot at time step $i-2$
S_{i-3}	original simulation snapshot at time step $i-2$
D_i	decompressed data at current time step i
D_{i-1}	decompressed data at time step $i-1$
D_{i-2}	decompressed data at time step $i-2$
D_{i-3}	decompressed data at time step $i-3$
P_i	predicted data at time step i
e	compression error bound
θ	impact factor (or impact bound ratio) defined in [28]

The impact-driven SDC detection (AID) method [28] uses a critical threshold called *impact factor* or *impact error bound ratio* (denoted as θ) to determine the nonnegligible data corruptions that would affect the execution results or post-analysis during the simulation. According to the AID detection [28], the impact of SDC (denoted by $I(t_i, t_{end})$) is defined as the maximum ratio of the absolute data change to the overall value range for the whole period the current time step i with SDC through the end of the execution (suppose an SDC happens to the current time step i). Then, the impact factor θ can be written as the the following formula according to [28], with an objective of limiting the impact of SDC under a predefined threshold φ.

$$\theta = \max_{I(t_i,t_{end}) \leq \varphi} \frac{\Delta_i}{r_i} \tag{4.2}$$

where Δ_i and r_i refer to the maximum data value change and the value range at time step i, respectively.

The impact factor θ is specified by users based on their applications in practice. For instance, the authors of AID [28] provided an in-depth analysis about the impact of SDC to the execution results, by comparing the fault-free execution results and SDC-injected execution results with 18 scientific applications across from different domains. They concluded that the impact factor could be set to 1E-3∼1E-4, under which the impact of SDC (i.e., $I(t_i, t_{end})$) could be bounded under 1% in most of cases.

4.4.1 Design Overview

For better understanding and easier explanation, Figure 4.4 illustrates three end-to-end SDC detection solutions designed for different execution modes (synchronous mode versus asynchronous execution) particularly. Basically, at each time step (i), AID reads the previous snapshot data (original historical data in synchronous mode or decompressed historical data in asynchronous mode) and performs the data prediction for each data point. We describe the three solutions briefly as follows, and present the details by pseudo-codes and discuss the pros and cons for each thereafter.

- *Solution A* is designed for the synchronous mode. It would compare the predicted data and snapshot data at time step i for detecting possible SDCs occurring during the simulation. Additionally, it compares the predicted data and decompressed data to verify if the decompressed data satisfies the user defined compression error bound, detecting potential errors happening during the compression. It is also called

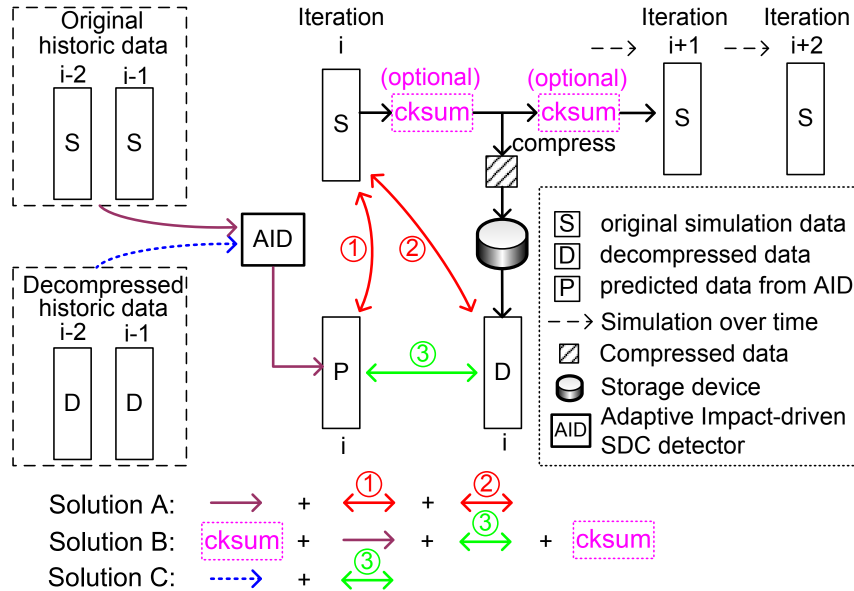


Figure 4.4: Design workflow of end-to-end SDC detectors at iteration i

synchronous end-to-end SDC detection with separate comparisons (SESD(S)) in the following text.

- *Solution B* is also designed for the synchronous mode. Unlike solution A, solution B combines the two comparison operations into one comparison between predicted data and decompressed data, in order to reduce the detection overhead as much as possible. This solution, however, needs to adopt checksum to protect the original simulation data against the possible SDCs happening during the compression period. It is named as synchronous end-to-end SDC detection with coupled comparisons (SESD(C)) in the following text.

- *Solution C* is designed particularly for the asynchronous mode. In this solution, the simulation would not be blocked by the SDC detection at all because of the totally asynchronous execution model. In this situation, it is impossible to compare the

original simulation dataset S_i and the predicted dataset P_i for detecting possible SDC errors, since the original simulation data are likely to be refreshed right away by the next simulation cycle (i+1) as the current simulation cycle ends. To address this limitation, the detection method will be performed asynchronously by a separate daemon (or one or more extra processes) and we can only compare the predicted data with the decompressed data to detect possible SDCs. We call this solution asynchronous end-to-end SDC detection (AESD) in the following text.

4.4.2 Impact Factor vs. Compression Error Bound

It is worth noting that the end-to-end detection involves two critical thresholds: impact factor θ and compression error bound (e), and the detection ability of the end-to-end SDC detection solutions is related to the mutual relationship between the two thresholds. In the rest of this subsection, we mainly discuss the impact factors and compression error bounds as well as their possible values.

Understanding Required Impact Factor

How to set an appropriate impact factor for a simulation depends on application datasets and user's target/objective. According to the previous solid experiments presented in [28], the required impact factor θ may span a large value range. That is, it could be very low or very high, depending on applications. Specifically, when the simulation is very sensitive to a tiny SDC, the impact factor has to be set to a small value. For instance, the impact (i.e., the maximum deviation of data values due to the SDC error propagation over time steps) could go up to 50%∼90% of the raw value range for the

BlastBS simulation [88] and Eddy simulation [78], as shown in [28]. By contrast, some applications are able to tolerate well relatively large SDCs on their own because of their stencil simulation nature, such that the impact factor could also be relatively large. A typical example is ConductionDelta [21], in which the impact is much less than 0.01% even if data value is changed 10% by SDC (i.e., impact factor = 10%) , as shown in [28].

Understanding Required Compression Error Bounds

Similar to the impact factor, the required compression error bound could also be either relatively large or very small, depending on the applications and user's post-analysis.

In some cases, the lossy compression could be performed under a relatively high compression error bound, still with an acceptable data distortion. According to climate researchers, visual quality is one critical indicator in their post-analysis. Figure 4.5 presents the visual quality of the FLDSC field (value range: [50,450]) in the CESM climate simulation under the SZ compression with different error bounds. It is clearly observed that the error bound of 1.0 can already lead to a very high visual quality even in a 1296× zoomed region, and other larger error bounds cause slight degradation on visual quality. The reason the visual quality is not degraded clearly with such a large error bound is that the compression errors actually follow a Gaussian-like distribution, in which most of the errors are near zero while the maximum value difference is close to the preset error bound, as shown in Figure 4.6.

As for some climate researchers' specific analysis [14], using a structural similarity index measure (SSIM) [80] (whose value is in the range of [0,1]) on the order of 0.99995

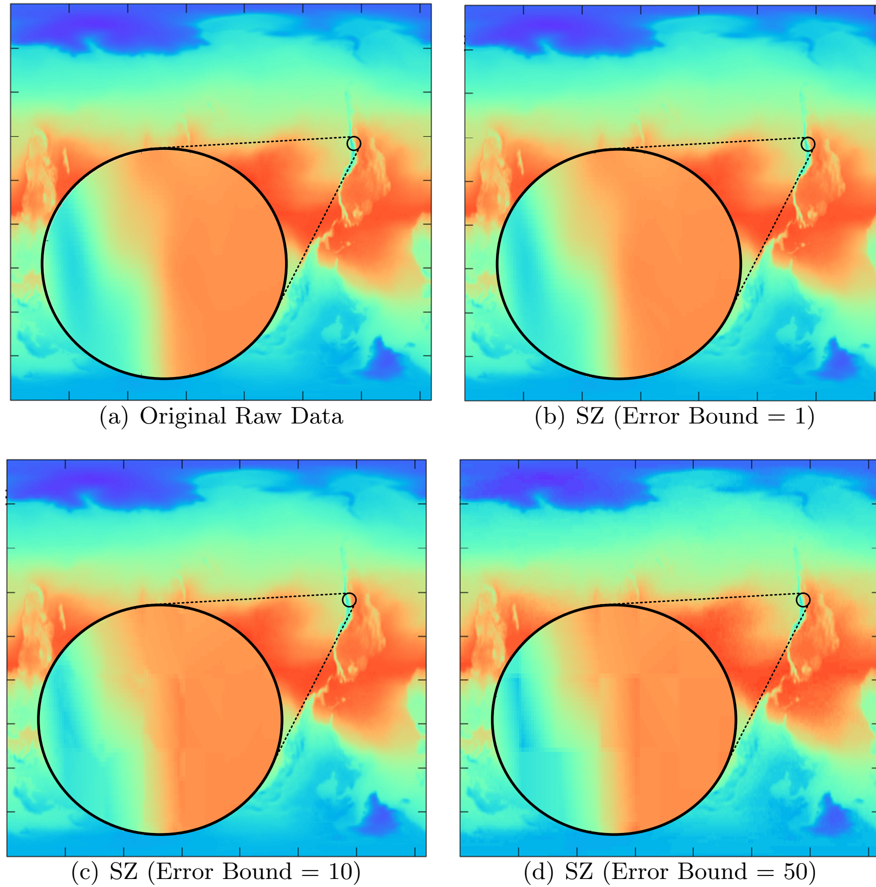


Figure 4.5: Visual Quality of Reconstructed Data with Different Compression Error Bounds (Climate Simulation CESM: FLDSC)

would be required. Such a high SSIM number corresponds to a fairly low compression error bound in general. Table 4.2, for example, presents the SSIM values of the reconstructed data (CESM FLDSC) under the error bounds of 1, 10, and 50 (corresponding to Figure 4.5 (b), (c) and (d)). We note that according to the SSIM criterion, none of the reconstructed data shown in Figure 4.5 are acceptable.

All in all, we can conclude that both impact factor and compression error bound could span a large value range, which totally depends on applications or user's requirement.

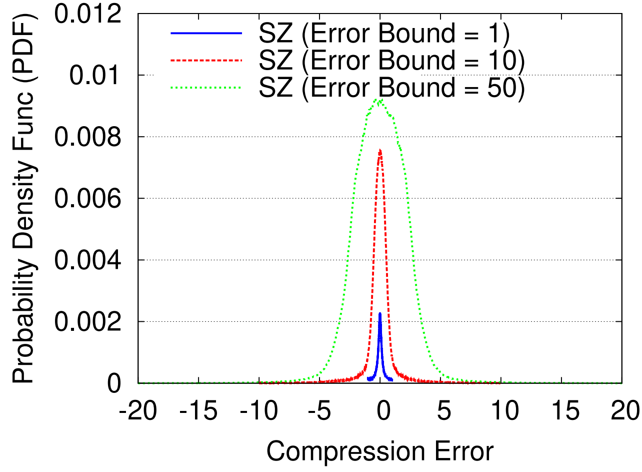


Figure 4.6: Distribution of Compression Errors

Table 4.2: SZ Compression Quality/Ratios of CESM FLDSC

Err Bound	**SSIM**	**PSNR**	**Max Err**	**Compre. Ratio**
1	0.99876	60.1 dB	1	97:1
10	0.99286	48.9 dB	9.99997	511:1
50	0.98754	43.8 dB	49.998	1545:1

So, we cannot assume that the compression error bound is generally smaller than the impact factor or vice verse. Instead, we need to take into account both situations in our designs.

4.4.3 Solution A: Synchronous End-to-End SDC Detection with Separate Comparisons (SESD(S))

Solution A is the most straight-forward solution, which adopts the traditional analytic-based SDC detection method (step ① in Figure 4.4) and verifies the correctness of the decompressed data (step ② in Figure 4.4), respectively. Algorithm 6 presents the corresponding pseudo-code, which is executed at each rank of the MPI parallel simulation (suppose the current time step is i).

Algorithm 6 SOLUTION A: SYNCHRONOUS END-TO-END SDC DETECTION WITH SEPARATE COMPARISONS (SESD(S))

Input: compression error bound e, snapshot S_{i-1}, S_{i-2}, S_{i-3}, impact factor θ
Output: SDC_detected = true or false

1: **for** (each variable at time step i) **do**
2: Compute value range r_i. /*to be used to compute ρ.*/
3: Select the bestfit prediction method [28].
4: **for** (each data point $s_j \in S_i$) **do**
5: Do prediction for s_j (denote the predicted value by p_j).
6: **if** ($|p_j - s_j| >$ confidence radius ρ_i) **then** /*check SDC*/
7: SDC_detected ← true.
8: break.
9: **end if**
10: **end for**
11: **if** (SDC_detected == true) **then**
12: Load checkpoint $i-1$ to redo simulation for time step i.
13: Verify if this is a false positive. /*rerun by checkpoint*/
14: **if** (the SDC is a false positive) **then**
15: Increase the confidence radius ρ based on Formula (4.3).
16: **end if**
17: **end if**
18: **end for**
19: Perform the error-bounded compression for S_i.
20: Write compressed snapshot data S'_i.
21: Read and decompress S'_i to get reconstructed dataset D_i.
22: **for** ($s_j \in S_i$ and $d_j \in D_i$) **do** /*Verify compression errors*/
23: **if** ($|s_j - d_j| > e$) **then**
24: SDC_detected ← true.
25: **end if**
26: **end for**

As shown in Algorithm 6, the SESD(S) first performs the regular AID detection algorithm [28] (line 1∼18). Specifically, it predicts each data point by its value in the three preceding time steps, and compares the snapshot dataset S_i versus the predicted data value P_i to detect potential SDCs.

$$\rho_i = (1 + \eta) \cdot (\varepsilon_{i-1} + \theta r_i) \qquad (4.3)$$

AID adopts an adjustable confidence radius (denoted by ρ_i) at time step i based on Formula (4.3), in order to minimize the false positives. In Formula 4.3, η refers to the number of false positive iterations up to the current time step i and ε_{i-1} is the maximum local prediction error in the last time step $i-1$. More details can be found in the paper [28].

When an outlier is detected by AID based on the smoothness of data (line 7), the simulation could rerun the corresponding simulation step based on the latest checkpointing file. If the same outlier is detected again, it would be marked as false positive and the confidence radius ρ will be increased (line 15) to avoid the possible future false positives. Such a design can adapt to the non-smooth data well during the simulation. Specifically, as the data behaves very spiky along time dimension, the confidence radius will increase automatically based on Formula (4.3), ensuring a low false positive despite a graceful degradation of detection ability. As shown in our prior work [28], based on the 18 real-world simulation, the detection sensitivity (i.e., the fraction of true alarms that are detected over all SDCs injected) is about 60% in the worst case, while the false positive rate stays around 1% (i.e., only 1 false alarm every 100 time steps) in most of cases.

After performing the AID SDC detection over the original snapshot data S_i, the SESD(S) compresses the snapshot data S_i (line 19) and writes the compressed data to the parallel file system (PFS) (line 20). After that, in order to guarantee the correctness of the written compressed data on PFS, we need to read and reconstruct the data for the compression error verification (line 22$\sim$26).

The pros and cons of this design will be discussed in detail later on, by comparing to the other solutions.

4.4.4 Solution B: Synchronous End-to-End SDC Detection with Coupled Comparisons (SESD(C))

In order to reduce the detection overhead, the solution B integrates the AID SDC detection and verification of compression errors together. To this end, this solution compares the predicted values with the decompressed data directly. This may suffer from lower detection ability (such as lower precision) while also getting lower execution overhead because of reduced operations.

Algorithm 7 SOLUTION B: SYNC. END-TO-END SDC DETECTION WITH COUPLED COMPARISONS (SESD(C))

Input: compression error bound e, snapshot S_{i-1}, S_{i-2}, S_{i-3}, impact factor θ
Output: SDC_detected = true or false

1: Set checksum for S_i.
2: $\rho \leftarrow \theta r_i$.
3: Perform the error bounded compression for S_i.
4: Write compressed snapshot data S'_i.
5: Read and decompress S'_i to get reconstructed dataset D_i.
6: Perform Algorithm 6's line 1∼18 with d_j replacing s_j. /*compare P_i versus D_i*/
7: Verify the correctness of S_i using checksum.

We present the pseudo-code in Algorithm 7. At the beginning, the solution sets the checksum for the original dataset (line 1) in case of the possible SDCs happening to it during the data compression and data writing phase. Then, our solution performs the error-bounded lossy compression on the simulation data. After performing the compression, writing, reading and decompression, we need to perform the detection operations (line 6), which is similar to line 1∼18 of Algorithm 1. At last, this solution needs to verify the correctness of original data based on the previously set checksum to guarantee they are SDC-free during the compression and data writing for next iteration in the simulation.

4.4.5 Solution C: Asynchronous End-to-End SDC Detection

The solution C is particularly designed for asynchronous I/O scenario. As shown in Figure 4.4, this solution involves the fewest steps, which just predicts the data by leveraging the decompressed data of preceding time steps ($i-1$, $i-2$, etc.) and compares it to the decompressed data at the current step i. The whole process is performed by an individual daemon (one or more additional processes) running in parallel with the simulation work.

We present the pseudo-code in Algorithm 3. As we mentioned before, the original simulation data of the current iteration may not be available to use in SDC detection if the data writing is not synchronized with the simulation execution. Moreover, the historical simulation data may not be available either because of the limited memory such that the original AID detector cannot use historical original data. As such, we can only use decompressed data (shown as blue dash line in Figure 4.4) to do SDC detection (i.e., line 6 in Algorithm 3).

Algorithm 8 SOLUTION C: ASYNC. END-TO-END SDC DETECTION (AESD)

Input: compression error bound e, snapshot D_{i-1}, D_{i-2}, D_{i-3}, impact factor θ
Output: SDC_detected = true or false

1: $\rho \leftarrow \theta r_i$.
2: Perform the error bounded compression for S_i.
3: Write compressed snapshot data S_i'.
4: Load decompressed data at preceding time steps (D_{i-1}, D_{i-2}, etc.) on demand.
5: Read and decompress S_i' to get reconstructed dataset D_i.
6: Perform Algorithm 6's line 1∼18 with d_j replacing s_j. /*compare P_i versus D_i*/

The key advantage of this solution is that it would not block the application's simulation work at all, because all the compression, data writing and SDC detection operations are conducted by a separate daemon (one or more additional processes) concurrently.

4.4.6 Inaccuracy that impacts SDC detection

Intuitively, our proposed end-to-end SDC detection solutions would be affected by the inaccuracy introduced when using the lossy compression. That is, lossy compression techniques may reduce the detection ability of the AID detector, which still depends on specific SDC detection methods. Specifically, solution A will not be impacted by compression error because AID uses original historical data and compares the predicted values with original data. The solution B will be affected by the lossy compression techniques less than solution C. The key reason is that the solution C predicts the data values based on the historical decompressed data instead of the original data, such that its data prediction may suffer from higher prediction errors. By contrast, the solution B still adopts the original historical data to perform the data prediction, which can keep the same detection ability in protecting the original simulation procedure with AID.

In the following, we analyze how much impact the compression error would have on the detection ability in absolute terms. Denote the original data points we want to detect by d, and denote the AID predicted value based on original history data by p. Given a scientific application, suppose the compression error bound is set to e, and the SDC impact factor is denoted as θ according to user's experience or analysis. That is, AID detects SDC by checking if $|p - d| > \theta$. The impact factor r could differ with different snapshots during the simulation, while the analysis stays the same. AID uses one of the three methods (see the following three formulas in Equation 4.4) to perform prediction: (1) last state fitting (LSF), (2) linear curve fitting (LCF), and (3) quadratic curve fitting (QCF) [28].

$$LSF : p_i = s_{i-1} \tag{4.4a}$$

$$LCF : p_i = 2s_{i-1} - s_{i-2} \tag{4.4b}$$

$$QCF : p_i = s_{i-1} - 3s_{i-2} + 3s_{i-3} \tag{4.4c}$$

Based on Equation (4.4), we can see that the maximum fluctuation for p will be at most $e + 3e + 3e = 7e$ when we are using QCF for prediction. For solution B, the inaccurate detection will become $|p - d'| > \theta$; for solution C, it will become $|p' - d'| > \theta$ where $|p' - p| \leq 7e$ and $|d' - d| \leq e$. So, the value of $|p - d|$ will deviate by at most e for solution B and deviate by at most $8e$ for solution C. As a result, the impact of the compression error on the detection ability of AID will depend on the difference between e and θ. Notice that the error bound e depends on user's demand on decompressed data quality and compression ratio. The impact factor θ depends on the simulation applications. If e is extremely smaller than θ, the impact of compression error on detection ability should be negligible. If e and θ have comparable values, the detection ability will be greatly impacted.

4.4.7 Overhead Analysis

The comparison base is the original simulation with added AID detector and lossy compressor. All the extra operations we added to make our solutions work in the context where lossy compression is used will be the overheads.

All three solutions we proposed inevitably have the overhead introduced by using lossy compression, reading compressed file and performing decompression. Since writing file is also a procedure in the base case, it is not counted as the overhead. Moreover, note that in Figure 4.4, any comparison between the AID predicted value should be considered as the original AID operations. For example, the comparison between P and S, the comparison between P and D in Figure 4.4. Thus, they are not considered as overheads either. With the above clarification in mind, we can derive the overhead for each of the three solutions as follows.

Without loss of generality, we assume the number of total data points in each snapshot is fixed (denoted by N). For solution A, the only added operations are comparing the decompressed data and original data to see if their differences are within the error bound. So, the time overhead will be at the order of N. For Solution B, the added operations are just two round of calculations for the checksums which are just $2n$. For solution C, the only change compared to the original design is that AID is applied on the decompressed data instead of original data which does not change the number of operations though. So the overhead for solution C will be zero. In terms of memory overhead, our design will not introduce significant amount of extra memory. Specifically, solution A and C will not introduce any extra memory usage. Solution B will have small constant of extra memory to hold the results of the checksums which is negligible.

Table 4.3: Basic info about scientific applications used in experiments

Simulations	# Snapshots	Dimensions	Science
Flash	500	64*64*64	Suite
Nek5000	300	104448	Suite
EXAALT	83	1077290	Chemistry
CESM	63	3600*1800	Climate

4.5 Evaluation and Discussion

4.5.1 Experimental Setup

To evaluate our proposed solutions, we perform the experiments on the Argonne Bebop supercomputer [1] using four widely used scientific simulations. Each node of this machine is driven by two Intel Xeon E5-2695 v4 processors with 128GB DRAM. The information about the applications is listed in Table 4.3. We choose the Sedov application from Flash suite and Vortex application from Nek5000 suite. We implemented the end-to-end SDC detection method by enabling the open-source AID package [28] to support two state-of-the-art lossy compressors (SZ and ZFP) and running the new solutions with the four scientific applications/datasets. For both of the compressors, all error bounds in this chapter are absolute error bounds. We measure average running time by repeating sequential program 10 times and parallel, 3 times. Compiler optimization -O3 is used for all programs.

4.5.2 Investigation of False Positives in Error-free Cases

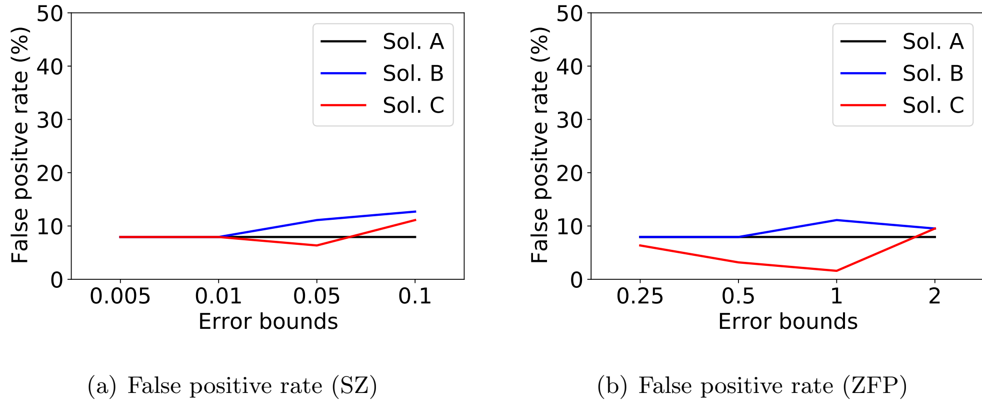
In this subsection, we characterize the execution time overheads and false positives for the three end-to-end SDC detection solutions, by running them with four different scientific simulations. Without loss of generality, we focus only on the situations with

relatively low false positives, which is the first high priority to applications. Otherwise, the applications cannot even advance because of too frequent roll-back operations (or restarting from checkpoints).

Detailed evaluation results are presented in Figure 4.7 through Figure 4.10. The baseline is the simulation equipped with some lossy compressor (SZ or ZFP), which is using AID to protect the original datasets but no protections for the duration of data compression and compressed data writing. Based on these figures, we can clearly observe that the false positive rate is determined by many factors, including specific application datasets, SDC solutions, particular lossy compressors, and also compression error bounds. The execution time overheads are observed always within 20% for all different settings.

We define the *acceptable compression error bound* as the compression error bound under which all the end-to-end SDC detection methods can get relatively low false positive rates [1]. By comparing Figure 4.7 (b) versus Figure 4.7 (d), it is observed that SZ has much smaller acceptable error bound than does ZFP. The key reason is that ZFP generally over-preserves the compression errors as verified by prior work [29]. For instance, if the error bound is set to 0.001 for both SZ and ZFP, the maximum compression errors after the compression/decompression is often much smaller than 0.001 (such as 0.0002) for ZFP, while it is very close to 0.001 for SZ in most of cases. This leads to much lower statistical errors such as mean squared errors for ZFP, which explains why the users need to choose a relatively small error bound for SZ in order to get relatively low false positives.

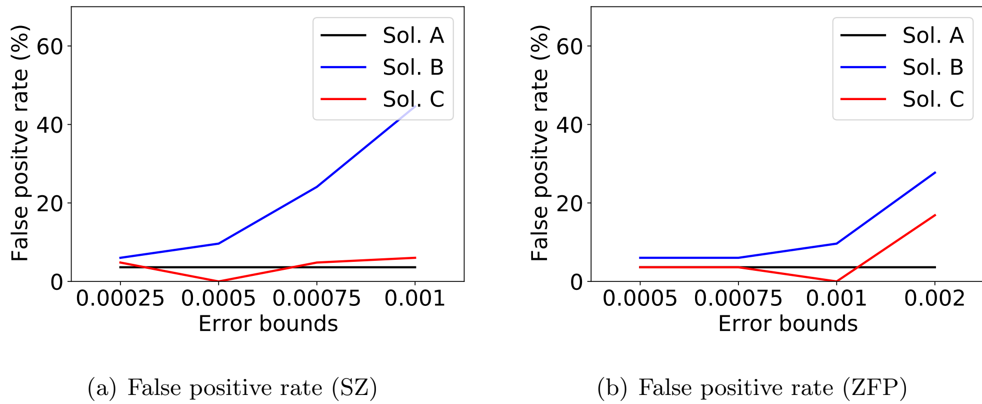
[1] false positive rate is defined as the fraction of the number of false positive time steps to the total number of time steps for some application



(a) False positive rate (SZ)

(b) False positive rate (ZFP)

Figure 4.7: False positive rate of CESM

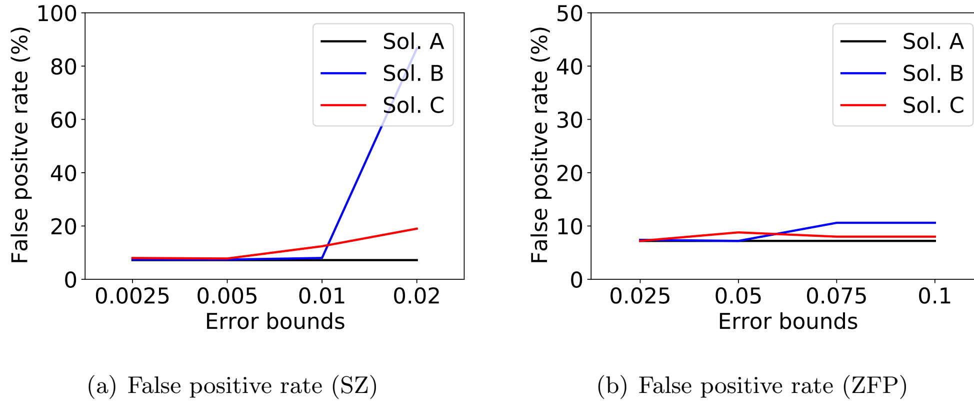


(a) False positive rate (SZ)

(b) False positive rate (ZFP)

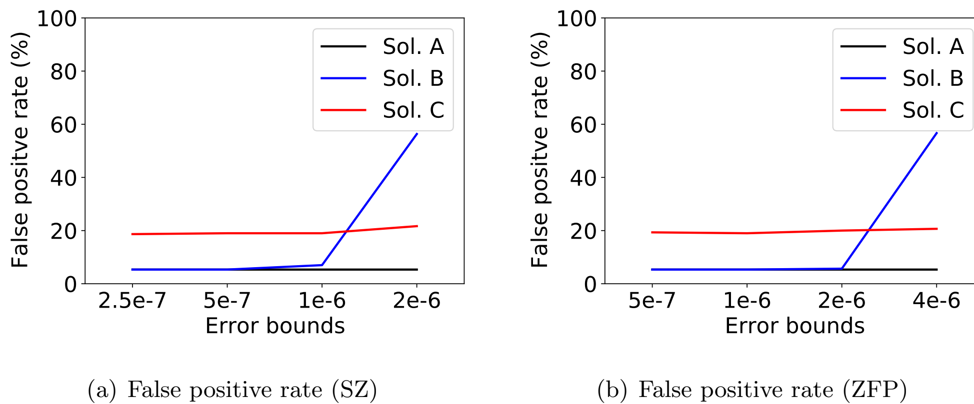
Figure 4.8: False positive rate on Exaalt

We summarize the acceptable error bounds and corresponding compression ratios of SZ and ZFP, based on all the four different scientific simulations. Quite a few studies [44, 51] have verified that the compression ratios can improve the simulation performance because of greatly reduced I/O time. However, it is unclear that how much the compression ratios could be applied on the applications, such that the end-to-end SDC detection could have a relatively low false positive rate. We summarize the results regarding SZ and ZFP in Table 4.4 and 4.5, respectively. As we mentioned previously, the false positive is related



(a) False positive rate (SZ) (b) False positive rate (ZFP)

Figure 4.9: False positives on Flash



(a) False positive rate (SZ) (b) False positive rate (ZFP)

Figure 4.10: Overheads and false positives on Nek5000

to compression error bounds, so we set error bounds for each dataset individually to make sure that the SDC detector still has acceptable false positives. Through the two tables, we can see that with comparable false positives, SZ has better compression ratios than ZFP on three datasets, Flash, Exaalt and CESM; while ZFP outperforms SZ on Nek5000 dataset. This motivates us to investigate the end-to-end SDC detection ability under both of the compressors in our analysis. The result suggests the following insights:

Table 4.4: Acceptable Error Bounds and Compression Ratios (SZ)

	Flash				Nek5000			
ebs:	0.0025	0.005	0.01	0.02	2.5e-7	5e-7	1e-6	2e-6
ratios:	139.53	162.80	187.66	216.83	6.25	6.42	7.04	8.22
	Exaalt				CESM			
ebs:	2.5E-4	5E-4	7.5E-4	0.001	0.005	0.01	0.05	0.1
ratios:	5.40	6.43	7.19	7.91	25.14	36.16	126.84	265.08

Table 4.5: Acceptable Error Bounds and Compression Ratios (ZFP)

	Flash				Nek5000			
ebs:	0.025	0.05	0.075	0.1	5e-7	1e-6	2e-6	4e-6
ratios:	72.61	80.85	90.71	90.71	11.82	12.38	13.01	13.70
	Exaalt				CESM			
ebs:	0.0005	0.00075	0.001	0.002	0.25	0.5	1	2
ratios:	2.57	2.57	2.78	3.03	26.08	29.33	32.99	37.80

- The false positive rate of Solution B and C increase with the selected error bounds.

- The false positive rate of solution B tends to increase faster than solution C with error bounds. This trend is obvious on Exaalt data. This observation is kind of contrast to the intuition, because the solution C involves higher compression errors in its design than the solution B, while solution C exhibits lower false positives. This can be explained as follows. According to Formula (4.2), the larger the prediction errors (i.e, ϵ_{i-1}) in the last time step, the larger the confidence radius is. Compared with the solution B, the solution C is supposed to have larger prediction errors because of the consistent data loss level between the decompressed data in last time step and the current time step, leading to larger confidence radius accordingly. So, solution C is more tolerable to the false positives, corresponding to lower false positive rates. Table 4.6 confirms our analysis: we clearly observe that solution B does have higher confidence radius than solution C, especially at the error bound, 0.1.

Table 4.6: Confidence Radius of Solution B and C on CESM with SZ at Time Step 40

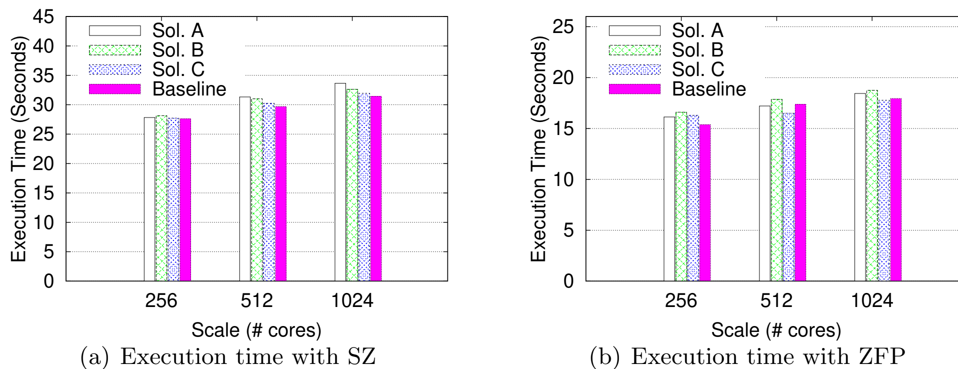
ebs	0.005	0.01	0.05	0.1
Solution B	0.658	0.658	0.658	0.658
Solution C	0.661	0.658	0.729	0.809

4.5.3 Investigation of Detection Performance in Erroneous Cases

Based on the acceptable error bounds, we run all the three end-to-end SDC detection solutions in the erroneous cases to check whether there are some SDCs that AID can detect but our proposed solutions cannot because of the impact of compression errors. We inject the errors in terms of real-world situation as thoroughly as possible. Specifically, we check all snapshots (i.e., time steps) one by one. For each snapshot, we randomly select one data point whose value will be modified by randomly flipping one bit such that its changed value is greater than the impact factor. In this way, we simulate only influential SDC errors that may affect the user's simulation results according to the definition of impact factor. For each dataset and each lossy compressor, we repeatedly injected random errors for 50 times. We observe that the false negatives are extremely rare for all the three detection solutions on all datasets for both compressors. For example, CESM data with solution C has only one false negative case in the largest error bound 0.1 when SZ is used. All other configurations do not have any false negative cases, because the impact bound and error bound are close to each other and the chance that flipping a bit will make the data fall between those two bounds is tiny.

4.5.4 Performance overheads in parallel environment

We evaluate the time overheads of all the three proposed solutions in a parallel environment with both SZ and ZFP lossy compressors based on the CESM data. We ran the experiments using 256∼1024 cores from Argonne Bebop supercomputer, and the results are presented in Figure 4.11. We use absolute error bound 0.05 for SZ and 1 for ZFP which are both the third column settings according to Table 4.4 and 4.5. We observe $\leq$ 7.9% execution time overhead in the parallel environment for all the three solutions compared with the baseline solution (AID + compression + data writing). Specifically, the overheads with SZ in parallel are at most 1.9%, 5.4% and 7.1% on 256, 512 and 1,024 cores, respectively. The overheads with ZFP in parallel are 7.9%, 2.8% and 4.4% using 256, 512 and 1,024 cores, respectively. The overall execution time increases with the execution scale because each rank keeps unchanged data size in our experiments, such that the total data size increases with the number of cores (weak scaling), leading to enhanced total time.



(a) Execution time with SZ

(b) Execution time with ZFP

Figure 4.11: Execution time in parallel

4.6 Related Work

To the best of our knowledge, none of the existing work studied how to protect HPC applications equipped with lossy compressors particularly against silent data corruptions (SDC). As such, we discuss the state-of-the-art SDC detection technologies generically designed for HPC applications.

From the perspective of hardware, error correcting code (ECC) has been implemented in memory to detect and correct bit flips in memory. In spite of its generality, the ECC protected memory usually can correct single bit flipped memory errors and can only detect 2 or more bit flipped errors with reasonable extra memory usage and the overhead will be increasing dramatically if ECC is used to detect and correct multiple bit flipped memory errors [62]. However, ECC cannot detect or correct computation error. A general remedy is modular redundancy which can double or triple the hardware or software execution. 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 SDCs has motivated algorithm based fault tolerance (ABFT). The main idea of ABFT is to exploit the special characteristics of an application or algorithm, such that they can be used to detect and correct soft errors. Despite the relatively high effort to put because of different charac-

teristics across various applications, the tiny overhead of ABFT compared with modular redundancy still makes it very promising. Most of the existing ABFT methods, however, focus on popular arithmetic algorithms such as matrix operations [35], convolution [89] and sorting [47]. To the best of our knowedge, no ABFT work has been done for lossy compression algorithms with the presence of soft errors.

Unlike the ABFT which is always designed for a particular HPC numerical algorithms, generic data-analytic based SDC fault tolerance (DBFT) method can be applied on any HPC simulations with multiple time steps. Because of its generality, DBFT is also very appealing to HPC scientific simulation community, though it may not obtain as high detection ability as ABFT does. Di et al. [27] proposed an efficient SDC detection method by leveraging a feedback control based prediction method and uniform sampling, which exhibits a high SDC detection ability. Thereafter, the authors proposed another outstanding DBFT based SDC detector, namely Adaptive Impact Driven (AID) SDC detector, which adopts an adaptive prediction method on different ranks of the running applications. The above two detection methods are both replying on the smoothness of the data in time dimension. Some other generic SDC detectors were also proposed by different researchers, while AID still generally exhibits excellent detection abilities in class. For instance, Subasi et al. [69] proposed a machine learning based detection method in terms of the AID detection model, which can lower the memory overhead to be nearly zero because it does not rely on the smoothness of data along time dimension. Wang et al. [79] proposed a deep learning based SDC detector, which can improve the detection ability to a certain extent, while suffering heavy computation overhead.

In this work, we focus on SDC detection for the HPC applications equipped with lossy compression techniques, as this is a significant gap to the modern applications nowadays.

4.7 Summary

In this chapter, we exploit a series of data-analytic based SDC detection techniques to detect any possible errors occurring from the beginning of the simulation work through the end of data writing at each time step, including all behaviors such as simulation, compression/decompression and I/O. We provide an in-depth analysis of the SDC detection ability of such end-to-end SDC detection techniques for both synchronous and asynchronous I/O scenarios. We evaluate all our proposed end-to-end SDC detection methods using 4 well-known scientific simulations with two state-of-the-art error-bounded lossy compressors, on both performance overhead and reliability of execution results.

Chapter 5

Conclusions

To conclude the thesis, we provide innovative algorithmic solutions to improve the compression ratio and resilience of lossy compressors. The improved lossy compressors then reduce I/O time and improve resilience for large scale scientific simulation applications. First, we show how the improved compression ratio can speedup the simulation execution at large scale. Then we improve the resilience of lossy compressors via algorithm-based fault tolerance (ABFT). Though ABFT takes effort in algorithm analysis and it lacks generality, our proposed fault tolerant lossy compression have shown their overwhelming advantage in terms of very low performance overheads over the traditional redundancy based fault tolerance. We also show how to efficiently provide resilient time based lossy compression by using efficient ABFT for sorting. Finally, our proposed end-to-end SDC detection for scientific simulations equipped with lossy compression has found promising insights for future deployment of even larger scale scientific simulations with SDC resilience.

We hope the thesis can inspire readers to think about the following future work.

- Resilience is not all on hardware. Software based resilience can be considered in some cases. Resilience can be done by hardware and software co-design. For example, can dedicated artificial intelligence (AI) accelerators achieve great fault tolerance with only software based resilience so that we can save some cost on hardware resilience part (such as ECC) and in the meantime reduce memory bandwidth requirement?
- Algorithm-based fault tolerance not only applies to arithmetic operations. It can be done efficiently for some discrete algorithms like sorting. So, are there any other important operations that we want to provide fault tolerance?
- The use cases of lossy compressors should not be limited to just storage or I/O part though this thesis focuses on this part. Brainstorm the use cases of lossy compression including communication, memory, checkpointing, AI models and so on.

Bibliography

[1] Bebop. https://www.lcrc.anl.gov/systems/resources/bebop/. Online.

[2] Berkeley lab checkpoint/restart (blcr) for linux. https://crd.lbl.gov/departments/computer-science/CLaSS/research/BLCR. Online.

[3] Blosc compressor. http://blosc.org. Online.

[4] Exaalt. https://www.exascaleproject.org/project/exaalt-molecular-dynamics-at-the-exascale-materials-science/. Online.

[5] Gzip. http://www.gzip.org. Online.

[6] New horizons: The first mission to the pluto system and the kuiper belt. nasa.gov/newhorizons. Online.

[7] Pds: The planetary data system. https://pds.jpl.nasa.gov. Online.

[8] Soft error. https://en.wikipedia.org/wiki/Soft_error. Online.

[9] Triple modular redundancy. https://en.wikipedia.org/wiki/Triple_modular_redundancy. Online.

[10] Zstandard. https://github.com/facebook/zstd/releases. Online.

[11] Laurent Alonso, Philippe Chassaing, Florent Gillet, Svante Janson, Edward M Reingold, and René Schott. Quicksort with unreliable comparisons: a probabilistic analysis. *Combinatorics, Probability and Computing*, 13(4-5):419–449, 2004.

[12] Cyrille Artho, Kuniyasu Suzaki, Masami Hagiya, Watcharin Leungwattanakit, Richard Potter, Eric Platon, Yoshinori Tanabe, Franz Weitl, and Mitsuharu Yamamoto. Using checkpointing and virtualization for fault injection. *International Journal of Networking and Computing*, 5(2):347–372, 2015.

[13] Jasjeet Singh Bagla. Cosmological n-body simulation: Techniques, scope and status. *Current science*, pages 1088–1100, 2005.

[14] Allison H Baker, Dorit M Hammerling, and Terece L Turton. Evaluating image quality measures to assess the impact of lossy data compression applied to climate simulation data. In *Computer Graphics Forum*, volume 38, pages 517–528. Wiley Online Library, 2019.

[15] Allison H Baker, Haiying Xu, John M Dennis, Michael N Levy, Doug Nychka, Sheri A Mickelson, Jim Edwards, Mariana Vertenstein, and Al Wegener. 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*, pages 203–214, 2014.

[16] L. Bautista-Gomez and F. Cappello. Exploiting spatial smoothness in hpc applications to detect silent data corruption. In *2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems*, pages 128–133, Aug 2015.

[17] Jon Bentley. *Programming pearls*, page 110. Addison-Wesley Professional, 2016.

[18] Jon L Bentley and M Douglas McIlroy. Engineering a sort function. *Software: Practice and Experience*, 23(11):1249–1265, 1993.

[19] Greg Bronevetsky, Daniel Marques, Keshav Pingali, and Paul Stodghill. Automated application-level checkpointing of mpi programs. In *ACM Sigplan Notices*, volume 38, pages 84–94. ACM, 2003.

[20] M. Burtscher and P. Ratanaworabhan. FPC: A high-speed compressor for double-precision floating-point data. *IEEE Transactions on Computers*, 58(1):18–31, Jan 2009.

[21] ASCF Center. Flash user's guide (version 4.2). `http://flash.uchicago.edu/site/flashcode/usersupport/flash2usersguide/docs/FLASH2.5/flash2ug.pdf`. Online.

[22] Chao Chen, Greg Eisenhauer, Matthew Wolf, and Santosh Pande. Ladr: low-cost application-level detector for reducing silent output corruptions. In *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*, pages 156–167, 2018.

[23] Jieyang Chen, Xin Liang, and Zizhong Chen. Online algorithm-based fault tolerance for cholesky decomposition on heterogeneous systems with gpus. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 993–1002. IEEE, 2016.

[24] Zhengzhang Chen, Seung Woo Son, William Hendrix, Ankit Agrawal, Wei-keng Liao, and Alok Choudhary. Numarck: machine learning algorithm for resiliency and checkpointing. In *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 733–744. IEEE, 2014.

[25] Hurricane ISABEL dataset. http://sciviscontest-staging.ieeevis.org/2004/data.html. Online.

[26] James Demmel and Hong Diep Nguyen. Fast reproducible floating-point summation. In *2013 IEEE 21st Symposium on Computer Arithmetic*, pages 163–172. IEEE, 2013.

[27] S. Di, E. Berrocal, and F. Cappello. An efficient silent data corruption detection method with error-feedback control and even sampling for hpc applications. In *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 271–280, May 2015.

[28] Sheng Di and Franck Cappello. Adaptive impact-driven detection of silent data corruption for hpc applications. *IEEE Transactions on Parallel and Distributed Systems*, 27(10):2809–2823, 2016.

[29] Sheng Di and Franck Cappello. Fast error-bounded lossy HPC data compression with SZ. In *2016 IEEE International Parallel and Distributed Processing Symposium*, pages 730–739. IEEE, 2016.

[30] Sheng Di and Franck Cappello. Optimization of error-bounded lossy compression for hard-to-compress hpc data. *IEEE transactions on parallel and distributed systems*, 29(1):129–143, 2017.

[31] David Jerome Fiala et al. Transparent resilience across the entire software stack for high-performance computing applications. 2015.

[32] Thomas E Fornek. Advanced photon source upgrade project preliminary design report. Technical report, Argonne National Laboratory (ANL)(United States). Funding organisation . . . , 2017.

[33] Al Geist. Supercomputing's monster in the closet. *IEEE Spectrum*, 53(3):30–35, 2016.

[34] Salman Habib, Vitali Morozov, Nicholas Frontiere, Hal Finkel, Adrian Pope, and Katrin Heitmann. Hacc: extreme scaling and performance across diverse architectures. In *SC'13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–10. IEEE, 2013.

[35] Kuang-Hua Huang and Jacob A Abraham. Algorithm-based fault tolerance for matrix operations. *IEEE transactions on computers*, 100(6):518–528, 1984.

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

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

[38] et al. Jeongnim Kim. QMCPACK: an open source ab initio quantum Monte Carlo package for the electronic structure of atoms, molecules and solids. *Journal of Physics: Condensed Matter*, 30(19):195901, 2018.

[39] JE Kay, C Deser, A Phillips, A Mai, C Hannay, G Strand, JM Arblaster, SC Bates, G Danabasoglu, J Edwards, et al. The Community Earth System Model (CESM) large ensemble project: A community resource for studying climate change in the presence of internal climate variability. *Bulletin of the American Meteorological Society*, 96(8):1333–1349, 2015.

[40] Gokcen Kestor, Burcu Ozcelik Mutlu, Joseph Manzano, Omer Subasi, Osman Unsal, and Sriram Krishnamoorthy. Comparative analysis of soft-error detection strategies: A case study with iterative methods. In *Proceedings of the 15th ACM International Conference on Computing Frontiers*, CF '18, page 173–182, New York, NY, USA, 2018. Association for Computing Machinery.

[41] Anand Kumar, Xingquan Zhu, Yi-Cheng Tu, and Sagar Pandit. Compression in molecular simulation datasets. In *International Conference on Intelligent Science and Big Data Engineering*, pages 22–29. Springer, 2013.

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

[43] G. Li, K. Pattabiraman, S. K. S. Hari, M. Sullivan, and T. Tsai. Modeling soft-error propagation in programs. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 27–38, June 2018.

[44] Sihuan Li, Sheng Di, Xin Liang, Zizhong Chen, and Franck Cappello. Optimizing lossy compression with adjacent snapshots for n-body simulation data. In *2018 IEEE International Conference on Big Data (Big Data)*, pages 428–437. IEEE, 2018.

[45] Sihuan Li, Sheng Di, Kai Zhao, Xin Liang, Zizhong Chen, and Franck Cappello. Sdc resilient error-bounded lossy compressor. *arXiv preprint arXiv:2010.03144*, 2020.

[46] Sihuan Li, Sheng Di, Kai Zhao, Xin Liang, Zizhong Chen, and Franck Cappello. Towards end-to-end sdc detection for hpc applications equipped with lossy compression. In *2020 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 326–336. IEEE, 2020.

[47] Sihuan Li, Hongbo Li, Xin Liang, Jieyang Chen, Elisabeth Giem, Kaiming Ouyang, Kai Zhao, Sheng Di, Franck Cappello, and Zizhong Chen. Ft-isort: efficient fault tolerance for introsort. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–17, 2019.

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

[49] Xin Liang, Sheng Di, Dingwen Tao, Zizhong Chen, and Franck Cappello. An efficient transformation scheme for lossy data compression with point-wise relative error bound. In *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 179–189. IEEE, 2018.

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

[51] Xin Liang, Sheng Di, Dingwen Tao, Sihuan Li, Bogdan Nicolae, Zizhong Chen, and Franck Cappello. Improving performance of data dumping with lossy compression for scientific simulation. In *2019 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 1–11. IEEE, 2019.

[52] S Lim. A fault tolerant parallel computing architecture for remote sensing satellites. 2009.

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

[54] Peter Lindstrom. Error distributions of lossy floating-point compressors. Technical report, Lawrence Livermore National Lab, 2017.

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

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

[57] Magnus Lundborg, Rossen Apostolov, Daniel Spångberg, Anders Gärdenäs, David van der Spoel, and Erik Lindahl. An efficient and extensible format, library, and api for binary trajectory data from molecular simulations. *Journal of computational chemistry*, 35(3):260–269, 2014.

[58] Gabriel Marcus, Zhirong Huang, Yuantao Ding, Tor Raubenheimer, Lanfa Wang, Marco Venturini, Paul Emma, and Ji Qiang. High fidelity start-to-end numerical particle simulations and performance studies for lcls-ii. 2015.

[59] Nor Rizuan Mat Noor and Tanya Vladimirova. 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)*, pages 115–122. IEEE, 2013.

[60] NYX simulation. `https://amrex-astro.github.io/Nyx`. Online.

[61] Andrey Omeltchenko, Timothy J Campbell, Rajiv K Kalia, Xinlian Liu, Aiichiro Nakano, and Priya Vashishta. Scalable i/o of large-scale molecular dynamics simulations: A data-compression algorithm. *Computer physics communications*, 131(1-2):78–85, 2000.

[62] Somnath Paul, Fang Cai, Xinmiao Zhang, and Swarup Bhunia. Reliability-driven ecc allocation for multiple bit error resilience in processor cache. *IEEE Transactions on Computers*, 60(1):20–34, 2010.

[63] Danny Perez, Luis Sandoval, Sophie Blondel, Brian D Wirth, Blas P Uberuaga, and Arthur F Voter. The mobility of small vacancy/helium complexes in tungsten and its impact on retention in fusion-relevant conditions. *Scientific Reports*, 7(1):1–9, 2017.

[64] James S Plank, Kai Li, and Michael A Puening. Diskless checkpointing. *IEEE Transactions on Parallel and Distributed Systems*, 9(10):972–986, 1998.

[65] T. Reza, J. Calhoun, K. Keipert, S. Di, and F. Cappello. Analyzing the performance and accuracy of lossy checkpointing on sub-iteration of nwchem. In *2019 IEEE/ACM 5th International Workshop on Data Analysis and Reduction for Big Scientific Data (DRBSD-5)*, pages 23–27, 2019.

[66] Abhishek Rhisheekesan, Reiley Jeyapaul, and Aviral Shrivastava. Control flow checking or not? (for soft errors). *ACM Trans. Embed. Comput. Syst.*, 18(1), February 2019.

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

[68] Daniel Spångberg, Daniel SD Larsson, and David van der Spoel. Trajectory ng: portable, compressed, general molecular dynamics trajectories. *Journal of molecular modeling*, 17(10):2669–2685, 2011.

[69] O. Subasi, S. Di, P. Balaprakash, O. Unsal, J. Labarta, A. Cristal, S. Krishnamoorthy, and F. Cappello. Macord: Online adaptive machine learning framework for silent error detection. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 717–724, Sep. 2017.

[70] ANL Theta supercomputer. `https://www.alcf.anl.gov/theta`. Online.

[71] Li Tan, Shashank Kothapalli, Longxiang Chen, Omar Hussaini, Ryan Bissiri, and Zizhong Chen. A survey of power and energy efficient techniques for high performance numerical linear algebra operations. *Parallel Computing*, 40(10):559–573, 2014.

[72] Dingwen Tao, Sheng Di, Zizhong Chen, and Franck Cappello. In-depth exploration of single-snapshot lossy compression techniques for n-body simulations. In *2017 IEEE International Conference on Big Data (Big Data)*, pages 486–493. IEEE, 2017.

[73] Dingwen Tao, Sheng Di, Zizhong Chen, and Franck Cappello. 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)*, pages 1129–1139. IEEE, 2017.

[74] Dingwen Tao, Sheng Di, Xin Liang, Zizhong Chen, and Franck Cappello. Improving performance of iterative methods by lossy checkponting. In *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*, pages 52–65, 2018.

[75] Dingwen Tao, Sheng Di, Xin Liang, Zizhong Chen, and Franck Cappello. Optimizing lossy compression rate-distortion from automatic online selection between sz and zfp. *IEEE Transactions on Parallel and Distributed Systems*, 30(8):1857–1871, 2019.

[76] Jiannan Tian, Sheng Di, Kai Zhao, Cody Rivera, Megan Hickman Fulp, Robert Underwood, Sian Jin, Xin Liang, Jon Calhoun, Dingwen Tao, et al. cusz: An efficient gpu-based error-bounded lossy compression framework for scientific data. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*, pages 3–15, 2020.

[77] BryanE Usevitch. Jpeg2000 compatible lossless coding of floating-point data. *EURASIP Journal on Image and Video Processing*, 2007(1):085385, 2007.

[78] Owen Walsh. Eddy solutions of the navier-stokes equations. *The Navier-Stokes Equations II – Theory and Numerical Methods*, pages 306–309, 1992.

[79] Chen Wang, Nikoli Dryden, Franck Cappello, and Marc Snir. Neural network based silent error detector. In *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 168–178. IEEE, 2018.

[80] Z. Wang and A. C. Bovik. Mean squared error: Love it or leave it? a new look at signal fidelity measures. *IEEE Signal Processing Magazine*, 26(1):98–117, Jan 2009.

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

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

[83] Panruo Wu, Qiang Guan, Nathan DeBardeleben, Sean Blanchard, Dingwen Tao, Xin Liang, Jieyang Chen, and Zizhong Chen. Towards practical algorithm based fault tolerance in dense linear algebra. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, pages 31–42, 2016.

[84] Panruo Wu, Dong Li, Zizhong Chen, Jeffrey S Vetter, and Sparsh Mittal. Algorithm-directed data placement in explicitly managed non-volatile memory. In *Proceedings of*

the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing, pages 141–152, 2016.

[85] Xin-Chuan Wu, Sheng Di, Emma Maitreyee Dasgupta, Franck Cappello, Hal Finkel, Yuri Alexeev, and Frederic T Chong. Full-state quantum circuit simulation by using data compression. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–24, 2019.

[86] Dow-Yung Yang, Ananth Grama, and Vivek Sarin. Bounded-error compression of particle data from hierarchical approximate methods. In *Proceedings of the 1999 ACM/IEEE conference on Supercomputing*, pages 71–es, 1999.

[87] Zheng Yuan, William Hendrix, Seung Woo Son, Christoph Federrath, Ankit Agrawal, Wei-keng Liao, and Alok Choudhary. Parallel implementation of lossy data compression for temporal data sets. In *2016 IEEE 23rd International Conference on High Performance Computing (HiPC)*, pages 62–71. IEEE, 2016.

[88] Andrew L. Zachary, Andrea. Malagoli, and Phillip. Colella. A higher-order godunov method for multidimensional ideal magnetohydrodynamics. *SIAM Journal on Scientific Computing*, 15(2):263–284, 1994.

[89] Kai Zhao, Sheng Di, Sihuan Li, Xin Liang, Yujia Zhai, Jieyang Chen, Kaiming Ouyang, Franck Cappello, and Zizhong Chen. Algorithm-based fault tolerance for convolutional neural networks. *arXiv preprint arXiv:2003.12203*, 2020.

[90] Kai Zhao, Sheng Di, Xin Liang, Sihuan Li, Dingwen Tao, Julie Bessac, Zizhong Chen, and Franck Cappello. Sdrbench: Scientific data reduction benchmark for lossy compressors.

[91] Kai Zhao, Sheng Di, Xin Liang, Sihuan Li, Dingwen Tao, Zizhong Chen, and Franck Cappello. Significantly improving lossy compression for hpc datasets with second-order prediction and parameter optimization. In *Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '20, page 89–100, New York, NY, USA, 2020. Association for Computing Machinery.

[92] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on information theory*, 23(3):337–343, 1977.