ADDRESSING FAULT TOLERANCE FOR STAGING BASED SCIENTIFIC WORKFLOWS

By

SHAOHUA DUAN

A dissertation submitted to the

Graduate School—New Brunswick

Rutgers, The State University of New Jersey

In partial fulfillment of the requirements

For the degree of

Doctor of Philosophy

Graduate Program in Computer Science

Written under the direction of

Manish Parashar

And approved by

New Brunswick, New Jersey

May, 2020

ABSTRACT OF THE DISSERTATION

Addressing Fault Tolerance for Staging Based Scientific Workflows

By SHAOHUA DUAN

Dissertation Director:
Manish Parashar

In-situ scientific workflows, i.e., executing the entire application workflows on the HPC system, have emerged as an attractive approach to address data-related challenges by moving computations closer to the data, and staging-based frameworks have been effectively used to support in-situ workflows at scale.

However, running in-situ scientific workflows on extreme-scale computing systems presents fault tolerance challenges which significantly affect the correctness and performance of workflows. First, scientific in-situ workflow requires sharing and moving data between coupled applications through data staging. As the data volumes and generate rates keep growing, the traditional data resilience approaches such as n-way replication and erasure codes become cost prohibitive, and data staging requires more scalable and efficient approach to support the data resilience. Second, Increasing scale is also expected to result in an increase in the rate of silent data corruption errors, which will impact both the correctness and performance of applications. Moreover, this impact is amplified in the case of in-situ workflows due to the dataflow between the component applications of the workflow. Third, since coupled applications in workflows frequently interact and exchange the large amount of data, simply applying the state of the art fault tolerance techniques such as checkpoint/restart to individual application component can not guarantee data consistency of workflows after failure recovery. Furthermore, naive use of these fault tolerance techniques to the entire workflows will limit the diversity of resilience approaches of application components, and finally incur a significant latency, storage overheads, and performance degradation.

This thesis addresses these challenges related to data resilience and fault tolerance for in-situ scientific workflows, and makes the following contributions. This thesis first presents CoREC, a scalable resilient in-memory data staging runtime for large-scale in-situ workflows. CoREC uses a novel hybrid approach that combines dynamic replication with erasure coding based on data access patterns. CoREC also provides multilevel data resilience to satisfy different fault tolerance requirements. Furthermore, CoREC introduces optimizations for load balancing and conflict avoiding encoding, and a low overhead, lazy data recovery scheme. Then, this thesis addresses silent error detection for extreme scale in-situ workflows, and presents a staging based error detection approach which leverages idle computation resource in data staging to enable timely detection and recovery from silent data corruption. This approach can effectively reduce the propagation of corrupted data and end-to-end workflow execution time in the presence of silent errors. Finally, this thesis addresses fail-stop failures for extreme scale in-situ scientific workflows, and presents a loose coupled checkpoint/restart with data logging framework for in-situ workflows. This proposed approach introduces a data logging mechanism in data staging which is composed by the queue based algorithm and user interface to provide a scalable and flexible fault tolerance scheme for in-situ workflows while still maintaining the data consistency and low resiliency cost. The research concepts and software prototypes have been evaluated using synthetic and real application workflows on production HPC systems.

Acknowledgments

First and foremost, I would like to thank my advisor, Dr. Manish Parashar, for his guidance and support throughout my doctoral study and research. I am grateful for his advice, encouragement, patience and cheerfulness during my years at Rutgers.

I would like to thank as well Dr. Santosh Nagarakatte, Dr. Sudarsun Kannan, and Dr. George Bosilca for serving as part of my committee members and taking time off their busy schedule to read and review my thesis.

I would like to give special thanks to the DataSpaces team, Tong Jin, Marc Gamell, Qian Sun, Pradeep Subedi, Philip Davis, and Zhe Wang, with whom I spent the best part of my years at Rutgers. Much of this work would not have been possible without their contributions.

I thank Dr. Keita Teranishi, Dr. Hemanth Kolla from Sandia National Laboratories, Dr. George Bosilca, Dr. Aurelien Bouteiller, and Dr. Barbara Chapman at CAARES team for their collaborations, valuable discussions and co-authoring research papers.

I would also like to thank my friends and colleagues at Rutgers Discovery Informatics Institute (RDI^2) for creating a collaborative, productive and motivating work environment.

I owe enormous thanks to my parents Guixiang Wang and Jianhe Duan, my wife Lu Sun, and my son Eilian Duan who have been a persistent source of love and encouragement for me.

Dedication

To my mother Guixiang Wang.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Background

Scientific workflows running on current and emerging extreme-scale systems are providing new opportunities for solving some of the most important problems in science and society, such as those being addressed by the US Exascale Computing Program (ECP) [56]. However, running such workflows at extreme scale presents significant challenges spanning all aspects of data management, such as data analysis challenge, data movement challenge, data storage challenge, and energy efficiency, etc. For example, the S3D [16] extreme scale scientific workflow, which is a coupled, multi-scale, multi-physics turbulent combustion workflow, involves intricate data-processing that includes multiple analyses performed at different temporal frequencies on non-overlapping subsets of data. To address the data-related challenges associated with coupled scientific workflows such as S3D executing at extreme scales, in-situ approaches based on data staging have emerged and are being used by applications on current high-end computing systems [23, 51]. Figure 1.1 illustrates an in-situ scientific workflow where the application components are coupled via an in-memory data staging framework. In this workflow, the primary scientific simulation is the data producer and the data consumer(s) include secondary simulations, analytics services, and/or visualization applications coupled to the data producer.

In-situ workflow approaches, such as those based on data staging and in-situ/in-transit data-management, have emerged as effective solutions for addressing data-related challenges at extreme scales, and are being adopted by applications across current high-end computing systems [23, 19, 45]. These techniques leverage resources (compute, storage) on the HPC system itself to support the data interaction and data couplings required by the workflows as well as to execute data-processing workflows close to where the data is being produced.

These techniques reduces the amount of data that needs to be moved off the system and stored to persistent storage (see Figure 1.1). For example, a multi-scale, multi-physics turbulent combustion application S3D [16], has an intricate data-processing workflow with multiple analyses performed at different temporal frequencies on non-overlapping subsets of data.

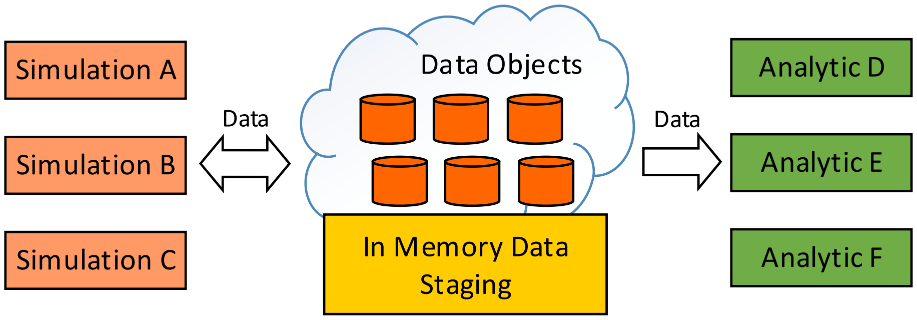


Figure 1.1: A typical data staging workflow.

DataSpaces [23], one such staging-based in-situ frameworks, uses data staging cores (cores on simulation and dedicated nodes for data staging) to implement a semantically specialized shared-space abstraction along with underlying runtime and RDMA-based asynchronous data transport mechanisms to effectively support in-situ/in-transit data analytics workflows requirements with minimal impact on the simulation itself [5, 46].

However, due to expected higher fault rates and dramatically increasing cost for addressing failures, there are several key fault tolerance challenges for building and effectively running in-situ scientific workflows on extreme-scale HPC systems. These challenges include the data reliability, data verification and crash consistency in in-situ workflows.

1.2 Research Challenges

1.2.1 Support for Data Reliability

To mitigate data loss on current and emerging extreme-scale systems which are expected to experience higher rates of failures [15], various fault tolerance techniques such as checkpoint/restart, process replication [26], and erasure coding [63] have been widely studied, and have been utilized. Unfortunately, current fault tolerance techniques can not be directly used to implement a resilient data staging services due to the contradiction between high memory and/or computation overhead for data resilience and limited resource with high performance requirement for data staging.

Even worse, data loss due to fail-stop failures (e.g., process failures, node failures) in any module of workflows, including data staging, will impact the execution of the entire workflow and can invalidate the final results. Consequently, it is critical to address data resilience of data staging for the extreme scales end-to-end workflow. **A data resilience mechanism that can support in-memory data recovery with high-performance, low overhead and minimum interference with regular data operation is needed to guarantee the data reliability for data staging.**

1.2.2 Support for Data Verification

Although various error-detection techniques for silent errors, such as ABFT [10] and time-series predictions [7], have been widely studied, these studies have generally been in the context of single applications rather than workflows, which are a composition of multiple interacting component applications. Moreover, addressing error detection for in-situ workflows including the analysis components is more complicated than single application's. The final results of the overall computation for the long-running extreme scale workflows are the outputs of the workflow, and silent errors in any component of the workflow can invalidate these outputs. Also, since the component applications of a scientific workflow exchange data, the impact of a silent errors will be propagated across application components in the workflow.

As a result, it is important to detect, isolate and correct silent errors in a component application as early as possible and to prevent the propagation of these errors between components. **A data verification mechanism is required for In-situ workflows to identify error corrupted components and minimize its impact.**

1.2.3 Support for Crash Consistency

Due to the dependencies, interactions and data exchanges between application components, naively applying states-of-the-arts fault tolerance scheme for the individual component can not maintain the consistent state of workflows during the failure recovery, which finally makes the result of workflows invalid or incorrect. Meanwhile, directly using uniform fault tolerance technique, such as global coordinated Checkpoint/Restart, to all application components to achieve crash consistency can make the complexity and overhead for addressing failures to be unacceptably high. Furthermore, this uniform fault tolerance strategy restricts the diversification for addressing application resiliency, and potentially increases the overall resiliency cost.

As a result, **it is requirement to design a loose coupled workflow-level fault tolerance mechanism to minimize the interference between components during recovery, while still maintaining consistent states of workflows.**

1.3 Overview of Thesis Research

The overall goal of this thesis is to address the research challenges related to data resilience, data verification, and crash consistency for in-situ scientific workflows, which are described above. This section presents an overview of thesis research.

This thesis presents CoREC (Combining Replication and Erasure Coding), which is a hybrid approach to data resilience for data staging. CoREC provides the benefit of data replication, i.e., high performance, while leveraging erasure coding to reduce storage costs. CoREC uses online data classification, based on spatial/temporal locality, to determine whether to use erasure coding or replication, and balances storage efficiency with low computation overheads while maintaining desired levels of fault tolerance. Moreover, to satisfy the diverse data resiliency requirements of different workflow components, CoREC supports the use of different data redundancy schemes, such as a hybrid approach combining both replication and triplication with different erasure codes. CoREC with multilevel data redundancy (CoREC-multilevel) can dynamically decide between erasure coding and replication schemes (for e.g., duplicate, triplicate, and Reed-Solomon codes) based on the data access patterns while maintaining application-specified resiliency requirement and incurring minimal storage overhead.

Furthermore, I develop a optimized load balance and conflict avoid encoding scheme for CoREC to decrease data encoding overhead further. To alleviate the overhead and interference associated with CoREC for both data-writes and data-recovery, I also develop a light overhead staging server recovery scheme based on lazy mode. I have used CoREC to implement resilient data staging within DataSpaces, and have also integrated ULFM (User Level Fault Migration) [8] to efficiently recover from both processes and node failures.

This thesis also explores the detection and remediation of silent errors for staging-based in-situ workflows. I leverage the fact that in-situ workflows exchange coupling data through data staging frameworks [21][24] and the idle computational resources within the staging area to provide uniform and efficient error detection. Specifically, I perform data validation as soon as the data is written to the staging area. If an error is identified in recently written data, the faulty application is instructed to roll back to the last known correct checkpoint and re-execute. In this way, the application waiting to consume the written data will only get access to the correct data, and error propagation from producer to consumer is eliminated. I use a spatial outlier detection method for data validation as an illustrative example in this thesis, and also explore how the performance impact of data verification can be reduced by leveraging GPGPU resources at the staging nodes to execute the error detection algorithms.

Finally, this thesis explores the workflow-level crash consistency strategy for in-situ workflows. I employ a data/event logging mechanism in the data staging to keep data consistency among application components during the failure recovery while decouple fault tolerance schemes between application components in workflows. Specifically, the data/event logging is performed as soon as the data is written or read through the staging area. Also, I introduce a global user interface to application components, which works with the queue based algorithm to record and replay data access events when preforming checkpointing and rollback recovery. In this way, our checkpoint/restart with data logging framework allows wide area fault tolerance schemes to be applied in workflows with flexibility and scalability, and minimize the interference between normal application components and the failed one when performing the recovery strategy.

1.4 Contributions

This thesis makes the following contributions.

- Design and implementation of CoREC, which provides scalable data resilience and process/nodes failure recovery for data staging. CoREC also enables multi-level data resilience with high performance and low overhead for the distributed in-memory data staging.
- Design and implementation of error detection framework in data staging, which provides data verification for staging based in-situ scientific workflows. By leveraging idle computation resource in staging, staging based error detection framework can effectively prevent error propagation among application components, and minimize the cost for recovering from silent errors.
- Design and implementation of checkpoint/restart with data logging framework for in-situ scientific workflows, which effectively keep data consistency during application failure recovery meanwhile providing a compatibility of diverse state-of-the-art fault tolerance approaches to individual application components.
- Implementations of the prototype within Open-sources data staging DataSpaces, and experimental evaluations using synthetic workloads and the S3D combustion workflow on the Titan Cray XK7 production system at Oak Ridge National Laboratory (ORNL), the Cori Cray XC40 system at the National Energy Research Scientific Computing Center (NERSC), and the Caliburn system at Rutgers Discovery Informatics Institute (RDI2).

1.5 Thesis Outline

The rest of this thesis is organized as follows.

Chapter 2 presents a high-level overview of the related fault tolerance approaches for realizing the resilient workflows.

Chapter 3 presents motivating and representative workflow examples, and summarizes the fault tolerance requirements.

Chapter 4 presents the model, design, implementation, and evaluation for CoREC.

Chapter 5 presents a staging based silent error detection framework.

Chapter 6 presents a workflow-level Checkpoint/Restart with data logging framework.

Chapter 7 summarizes the research work, presents concluding remarks and directions for future work.

Chapter 2
Background and Related Work

There are many types of errors, faults, or failures. Some are transient, others are unrecoverable. Some cause a fatal interruption of the application as soon as they strike, others may corrupt the data in a silent way and will manifest only after an arbitrarily long delay.

While there is an increasing body of work on scalable fault-tolerance mechanisms applicable to individual applications [31], these mechanisms are not directly applicable to in-situ scientific workflows, and to the data staging service supporting the workflow and the data being staged by the workflow. To tackle this problem, this dissertation mainly focuses on fail-stop failures and silent errors, and address data resilience challenges which are related with these two type of failures for in-situ scientific workflows. In this chapter, we first investigate failure rate of current and future extreme scale system. We then explore why traditional mechanisms, such as Checkpoint/Restart, are unable to effectively meet resilience requirements of staging-based in-situ workflows.

2.1 Failures in Extreme Scale HPC Systems

For HPC application workflows, scale is a major opportunity. Massive parallelism with 100,000+ nodes is the most viable path to achieving sustained Petascale performance. Future platforms will enroll even more computing resources to enter the Exascale era. Current plans refer to systems either with 100,000 nodes, each equipped with 10,000 cores (the fat node scenario), or with 1,000,000 nodes, each equipped with 1,000 cores (the slim node scenario)[37].

Unfortunately, scale is also a major threat, because resilience becomes a big challenge. Even if each node provides an individual MTBF (Mean Time Between Failures) of, say, one century, a machine with 100,000 such nodes will encounter a failure every 9 hours in average, which is larger than the execution time of many HPC applications. Worse, a machine with 1,000,000 nodes (also with a one-century MTBF) will encounter a failure every 53 min in average.1 Note that a one-century MTBF per node is an optimistic figure, given that each node is composed of several hundreds or thousands of cores.

To further darken the picture, everything else is not equal: smaller transistors are more error prone. One major cause for transient hardware errors is cosmic radiation. High-energy neutrons occasionally interact with the silicon die, creating a secondary cascade of charged particles. These can create current pulses that change values stored in DRAM or values produced by combinatorial logic. Smaller circuits are more easily upset because they carry smaller charges. Furthermore, multiple upsets become more likely. Smaller feature sizes also result in larger manufacturing variances, hence larger variances in the properties of transistors, which can result in occasional incorrect or inconsistent behavior. Smaller transistors and wires will also age more rapidly and more unevenly so that permanent failures will become more frequent. Energy consumption is another major bottleneck for exascale. Subthreshold logic significantly reduces current leakage but also increases the probability of faults.

Vendors can mitigate for the increase in fault rates with various techniques. For example, for regular memory arrays, one can use more powerful error correction codes and interleave coding blocks in order to reduce the likelihood of multiple bit errors in the same block. Buses are usually protected by using parity codes for error detection and by retries; it is relatively easy to use more powerful codes. Logic units that transform values can be protected by adding redundancy in the circuits. Researchers have estimated that an increase in the frequency of errors can be avoided at the expense of 20% more circuits and more energy consumption [49]. Whether such solutions will be pursued is unclear, however: the IC market is driven by mobile devices that are cost and energy sensitive and do not require high reliability levels. Most cloud applications are also cost sensitive but can tolerate higher error rates for individual components. The small market of high-end servers that require high reliability can be served by more costly solutions such as duplication or triplication of the transactions. This market is not growing in size or in the size of the systems used. Thus, if exascale systems will be built out of commodity components aimed at large markets, they are likely to have more frequent hardware errors that are not masked or not detected by hardware or software.

As hardware becomes more complex (heterogeneous cores, deep memory hierarchies, complex topologies, etc.), system software will become more complex and hence more error-prone. Failure and energy management also add complexity. Similarly, the increase use of open source layers means less coordinated design in software, which will increase the potential for software errors. In addition, the larger scale will add complexities as more services need to be decentralized, and complex failure modes that are rare and ignored today will become more prevalent.

Application codes are also becoming more complex. Multiphysics and multiscale codes couple an increasingly large number of distinct modules. Data assimilation, simulation, and analysis are coupled into increasingly complex workflows. Furthermore, the need to reduce communication, allow asynchrony, and tolerate failures results in more complex algorithms. Like system software, these more complex algorithms and application codes are more error-prone.

Researchers have predicted that large parallel jobs may fail as frequently as once every 30 minutes on exascale platforms [49]. Such failure rates will require new error-handling techniques. Furthermore, silent hardware errors may occur, requiring new error-detection techniques in (system and/or application) software.

2.2 Data Resilience Techniques

2.2.1 Data Replication

Since ensuring access to the staged data in spite of failures is most critical for a staging service, data resilience techniques such as data replication or erasure coding are more appropriate. Traditional ways of providing data reliability are through replication, by which a dataset is replicated $M+1$ times to tolerate M failures. Actually, replication schemes such as standard primary-backup (PBR) [14] and chain-replication [48] have been widely used for building highly available in-memory KV-store systems. For example, large-scale in-memory key/value systems like Memcached [30] and Redis [62] have performed in-memory replication scheme to provide both high throughput and high availability so that such systems can continuously handle millions of requests per second in presence of frequent fail-stop failures. However, this also means dedicating M copies of CPU/memory without producing user work, and requiring more storage requirements, standby machines and thus multiplying energy consumption. For example, tolerating two node failures requires at least 200% storage overhead, which may not be feasible for in-memory staging due to limited memory size and staging nodes.

2.2.2 Erasure Codes

An alternate approach to data resilience with lower storage overheads is to employ erasure coding techniques. Erasure codes are constructed using two configurable parameters n and k (where $k < n$). The data is treated as a collection of fixed size units called blocks/objects. Every k original objects (called data objects) are encoded into $n - k$ additional equal size coded objects (called parities) and the set of the n data and parity objects is called a stripe. In case of a data staging service, objects of independently encoded multiple stripes are stored on distinct staging servers, allowing the service to tolerate $n - k$ server failures.

The motivation for using erasure coding comes from the need to reduce the cost of storage. Erasure coding can reduce the cost of storage over 50%, and is widely used as off-line data resilience with a tremendous cost saving in an Exabyte of storage. One typical use case of erasure coding is to provide data reliability for log file system. In the log system, data is appended to the end of active log file, which are replicated to keep the data durable. Once reaching a certain size (e.g., 1 GB), the log files are sealed. These sealed log files can no longer be modified and thus make perfect candidates for performing erasure coding, and once the data is erasure-coded the original replicas of the log files are deleted.

The trade-off for using erasure coding instead of replication is performance and availability. The data in the failed data staging server will be offline and not available during performing decoding for data recovery. Also, while erasure coding provides lower storage overheads as compared to the replication, it can lead to significant computation and network overheads as parity has to be re-computed for every object update. If a data object in a stripe is updated, erasure coding must update the associated parity. This process involves reading old data objects in the stripe, re-computing parities and updating them. For example, if a stripe has 6 data objects and 2 parity objects, updating one data object requires 5 data object reads (for old data), re-computing 2 parity objects and 2 parity object writes. As a result, using erasure coding can be suboptimal for frequently written/updated data objects.

2.3 Checkpoint/Restart

2.3.1 Coordinated Checkpointing

Checkpoint/Restart (C/R) is the most widely used fault-tolerant technique for recovering fail-stop failures in high performance applications. The principle of strategy for (C/R) is: checkpoints are periodically saved during application initial execution, and when processes are subject to failures, it uses these checkpoints to rollback the application processes to the latest consistent state, and re-execute the program from that point. Using Checkpoint/Restart for fault tolerance of the data staging service presents two concerns. The first is the impact on the runtime of application workflows that use the data staging service.

To illustrate this impact, we performed periodic checkpointing of the data stored at the DataSpaces servers to the parallel file system on Titan and measured the total execution time with no server failure. Checkpointing was performed every 5 minutes, which is similar to the frequency used in the experiments presented in [31], for a total of 8 staging servers with varying staged data sizes. This resulted in a range from 17 checkpoints for a data size of 4G to 20 checkpoints for a data size of 32G. The results are plotted in Figure 2.1. From the plots, we can see that even if no failures are present, checkpointing significantly increases the total execution time of the workflow as the staged data size increases. In this case, the maximum time spent to achieve fault tolerance for just the staging servers is $\sim 15.6\%$ of the workflow run-time without failures. In addition, this does not include the work lost from rolling back to a previous state. As presented in the chapter 4, failure recovery using CoREC increases the total execution time of the workflow by up to 2.23%, which is significantly

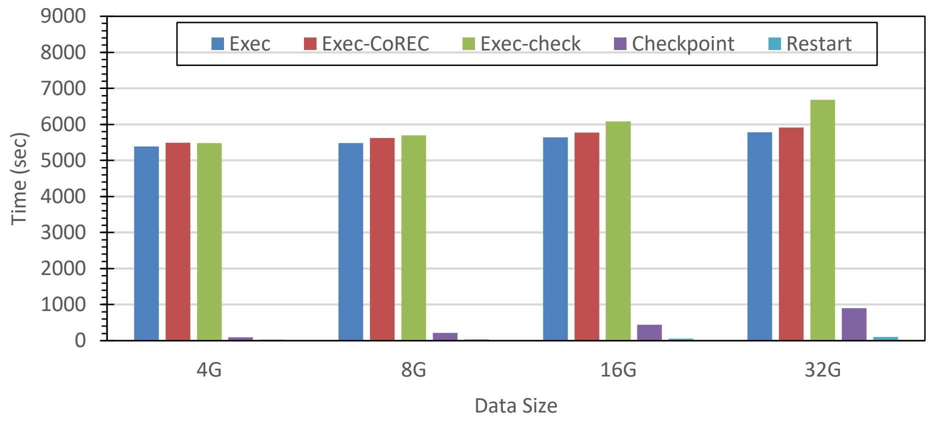


Figure 2.1: Impact of checkpointing on staging-based in-situ application workflows. *Exec* is the total execution time of the workflow without checkpointing; *Exec-CoREC* is the total execution time of the workflow using CoREC; *Exec-check* is the total execution time of the workflow with periodic checkpointing of the staged data; *Checkpoint* is the total time required to checkpoint the data staging servers; *Restart* is the time required to perform a global restart of the data staging servers using a checkpoint.

lower than using Checkpoint/Restart. Furthermore, there is no loss of work in the case of CoREC. The second concern is the overhead due to large amounts of data movement and potential cascading rollback. When using Checkpoint/Restart for MPI applications, rolling back the data staging server can cause the tightly coupled application components of the workflow to become out of sync. Since all of the tightly coupled components in the MPI communication group must be rolled back to an overall consistent state, this can trigger a cascading rollback of the workflow where the rollback of one component triggers other healthy component(s) to rollback, and, in the worst case, cause the entire workflow to restart from the beginning. This process can result in significant coordination and data movement overheads.

Uncoordinated Checkpointing with Message Logging

The coordinated checkpointing require that all processes rollback to the last valid checkpoint wave, when a failure occurs. This ensures a global consistency, at the cost of scalability: as the size of the system grows, the probability of failures increase, and the minimal cost to handle such failures also increase.

To reduce the inherent costs of coordinated checkpointing, uncoordinated checkpointing with message logging have thus been proposed. On the failure-free part of the execution, the main idea is to remove the coordination of checkpointing, targeting a reduction of the I/O pressure when checkpoints are stored on shared space, and the reduction of delays or increased network usage when coordinating the checkpoints. Furthermore, uncoordinated protocols aim at forcing the restart of a minimal set of processes when a failure happens. Ideally, only the processes subject to a failure should be restarted.

In uncoordinated checkpointing with message logging, processes log nondeterministic events and message payloads as they proceed along the initial execution; without strong coordination, they checkpoint their state independently; in case of failure, the failed process restarts from its last checkpoint, it collects all its log history, and enters the replay mode. Replay consists in following the log history, enforcing all nondeterministic events to produce the same effect they had during the initial execution. Message payloads must be provided to this process for this purpose. If multiple failures happen, the multiple replaying processes may have to reproduce the messages to provide the payload for other replaying processes, but since they follow the path determined by the log history, these messages, and their contents, will be regenerated as any deterministic action. Once the history has been entirely replayed, by the piecewise deterministic assumption, the process reaches a state that is compatible with the state of the distributed application, that can continue its progress from this point on.

Uncoordinated checkpointing with message logging has been explored for efficiently minimizing application vulnerability to fail-stop failures, and seems to an option for addressing fail-stop failures in in-situ workflows. Unfortunately, to the best of our knowledge, there is not a supported framework to allow an uncoordinated checkpointing mechanism running at in-situ workflows level and maintaining a global consistency.

2.4 Data Verification

Silent error is another potential threat to the data integrity of an HPC application. Silent errors may occur in the form of transient bit-flips and are typically caused by electronic noise or strikes by high energy particles, such as cosmic rays or proton radiation. Although the mean time between silent errors (which we will denote as MTBE in this thesis) of individual components is high, at extreme scales the aggregate MTBE of the entire system is low due to the large number of system components. Additionally, silent errors are becoming more prevalent in HPC systems as lower power chips with smaller feature sizes are being deployed. Research has shown that there exists a strong inverse correlation between the spontaneous error rate and the device sizes and operating voltages [28]. Compounding the issue, scientific workflows, due to their complexity and long execution times, tend to encounter interruptions or obtain invalid results caused by silent errors at a relatively high rate. Therefore, data resiliency is a critical concern for scientific workflows running on extreme scale systems and an efficient silent errors detection approaches is important for guaranteeing data resilience.

In contrast to a fail-stop failure whose detection is immediate, a silent error is identified only when the corrupted data leads to an unusual application behavior. Such a detection latency raises a new challenge: if the error struck before the last checkpoint, and is detected after that checkpoint, then the checkpoint is corrupted and cannot be used for rollback. In order to avoid corrupted checkpoints, an effective approach consists in employing some verification mechanisms and combining it with checkpointing. This verification mechanism can be general-purpose (e.g., based on replication or even triplication [29]) or application-specific (e.g., based on Algorithm-based fault tolerance (ABFT) [10], or on data dynamic monitoring [7]). In this section, we will briefly talk about three widely-used error detection approaches in application-level and figure out generality, accuracy and performance of each approach. The accuracy of error detection is quantified by using two measures: *recall* and *precision*, which are defined as:

$$recall = \frac{TruePositives}{TruePositives + FalseNegatives}$$

$$precision = \frac{TruePositives}{TruePositives + FalsePositives}$$

2.4.1 Process Replication

Process replication [29] has been widely used for tolerating both silent errors and fail-stop failures. It creates replica tasks, such as MPI process, for each primary task and compares the computation results between replicated processes to detect corrupted data. This approach is very general with high *recall* and *precision.* Unfortunately, process replication may not always be feasible for extreme scale in-situ workflows because computing redundant tasks of any components including data staging will impose large computation and storage overheads which significantly degrade the overall performance of in-situ workflows, since application components such as scientific simulation are usually computation intensive, and the compute resources in data staging are much smaller than that in that application components.

Also, using replication for fault tolerance of the data staging service would require each staging server and its data to be replicated, which doubles the compute and storage requirements and can make it infeasible.

2.4.2 ABFT

The general idea of Algorithm Based Fault Tolerance (ABFT) [10] is to introduce information redundancy in the data, and maintain this redundancy during the computation. Linear algebra operations over matrices are well suited to apply such a scheme: the matrix (original data of the user) can be extended by a number of columns, in which checksums over the rows are stored. The operation applied over the initial matrix can then be extended to apply at the same time over the initial matrix and its extended columns, maintaining the checksum relation between data in a row and the corresponding checksum column(s). Usually, it is sufficient to extend the scope of the operation to the checksum rows, although in some cases the operation must be redefined.

If a failure hits processes during the computation, the data host by these processes is loss or corrupted. However, in theory, the checksum relation being preserved, if enough information survived the failure between the initial data held by the surviving processes and the checksum columns, a simple inversion of the checksum function is sufficient to reconstruct the missing data or detect the silent errors.

No periodical checkpoint is necessary, and more importantly the recovery procedure brings back the missing or corrupted data at the point of failure, without introducing a period of re-execution as the general techniques seen above impose, and a computational cost that is usually linear with the size of the data. Thus, the overheads due to ABFT are expected to be significantly lower than those due to rollback-recovery.

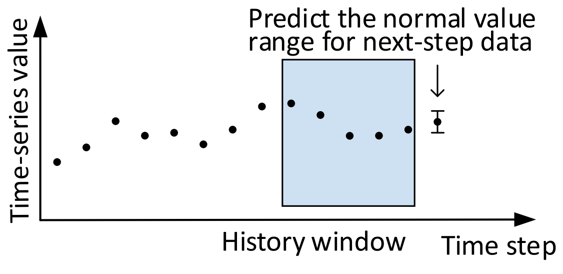
ABFT was shown to be an effective method with low computation overhead and high accuracy for application-layer detection and correction for a range of basic matrix operations including addition, multiplication, scalar product, transposition. Also, such techniques were also proven effective for LU factorization, Cholesky factorization and QR factorization. However, ABFT does not provide general error detection for applications. It has only been implemented for a limited set of linear algebra application, and so is only available to a small subset of the vast spectrum of scientific applications.

2.4.3 Outlier Based Error Detection

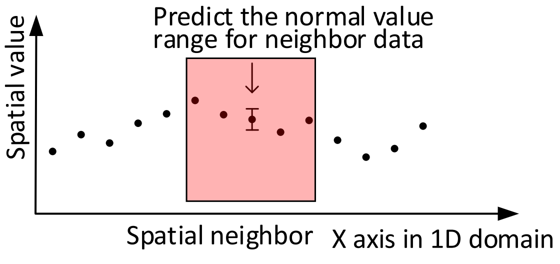
Most of the datasets produced by HPC scientific applications have expected informational characteristics that reflect the properties of the underlining physical phenomena that the applications seek to model [33]. Silent data corruption via bit flips alters the value of the data causing it to deviate from these standard characteristics. Therefore, corrupted data can be detected as an outlier based on its deviation from the expected range of normal values. There are two classes of outlier detection: time-series outlier detection and spatial outlier detection. Time-series outlier detection [7] is illustrated in Figure 2.2(a). HPC scientific application often iteratively operate upon data, changing their values over time. At each iterative time-step, a time-series outlier detection approach can be used to dynamically predict the possible range for data values at the next time-step, and a data value can be considered as an outlier if it falls outside this range. Modeling the value of a data point will take with a random variable, it is then possible to find outliers using variations of the Chebyshev's inequality,

$$P(|X - \mu| \geq k\delta) \leq \frac{1}{k^2}$$

where μ and δ are the mean and variance of the random variable X that models the data point be considered. For example, data value can be identified as silent errors when it cannot hold this inequality. When additional information is available, like the distributional assumption of X , this inequality can be sharpened. For example, when X follows a normal distribution, it can be shown that 99.7% of the data lies between three standard deviations, as opposed to 88.8% given by the general Chebyshev's inequality.



(a) Time-series outlier detection



(b) Spatial outlier detection

Figure 2.2: Examples of time series and spatial outlier detection techniques for a 1D data domain.

Spatial outlier detection [3] is illustrated in Figure 2.2(b). A spatial outlier is a spatially-referenced object whose non-spatial attribute values are significantly different from those of other spatially referenced objects in its spatial neighborhood. A spatial neighborhood may be defined based on spatial attributes, e.g., location, using spatial relationships such as distance or adjacency. Non-spatial attributes between spatially referenced objects can be compared within that neighborhood, considering those values whose non-spatial attributes deviate significantly to be spatial outliers. Although not perfectly accurate, outlier detection techniques can still detect a substantial fraction of silent errors, and more importantly these methods incur low overhead. These properties make them attractive candidates for staging-based error detection framework which will be discussed in chapter 5

Although outlier-based error detection approaches are good candidates for SDC detection, it should be noted that the choice of the most appropriate error detection approach depends on the data characteristics of the workflow; this thesis aims to provide a pluggable approach for enabling error detection in the staging area and to highlight its benefits, and can use the chosen most appropriate error detection approach for a specific workflow.

2.5 Summary

This chapter described the fail-stop failures and silent errors on extreme-scale computing systems, and presented five widely-used data and process resilience approaches, and figured out why directly using these state-of-the-art fault tolerance approaches in in-situ scientific workflows can not address fault tolerance related challenges efficiently.

Chapter 3

Motivating Applications and Challenges

Chapter 2 described the failures and state-of-the-art approaches for addressing them in HPC system. This chapter presents an overview of the technical approaches and implementations used for in-situ scientific workflows. In addition, this chapter describes the underlying technology, i.e., DataSpaces framework, which is used by the implementations presented in this thesis.

3.1 Motivating Staging Based In-situ Scientific Workflows

3.1.1 Coupled Combustion Simulation DNS-LES Workflow

S3D is a massively parallel computational fluid dynamics (CFD) solver that performs first principles based"direct numerical simulations"(DNS) of turbulent combustion. DNS is very expensive both in terms of flops and data generation, since it resolves the entire range of spatial and temporal scales in the continuum regime of given problem. Another paradigm of turbulent combustion simulations that is less expensive and more suitable for engineering calculations is "large eddy simulations" (LES), which only resolves the large energy containing a range of scales and models the physics for smaller scales. Coupled DNS-LES simulations are being considered as a rigorous, albeit expensive, test bed for assessing models, because they are capable of isolating and eliminating numerical errors. This is achieved by performing DNS and LES in lockstep, where the base solution field from the well resolved DNS solution is appropriately filtered and then fed to the LES simulation, which is running in tandem and is solving only one additional quantity whose model is being tested.

The lock-step DNS-LES coupling requirement presents a number of data management challenges. The simulations advance in time a six-stage Runge-Kutta scheme, which implies that the exchange of data between DNS and LES must happen six times every time step, and each time step typically requires a few seconds of wall-clock time. This effectively means frequent exchanges of a large volume of data. However, due to several uncertainties during execution (e.g., lag of LES simulation, network issues while transferring the data, etc.), the DNS and LES simulations often go out of sync. As a result, a large amount of data has to be cached over multiple time steps before the lock-step data exchange can recover.

3.1.2 Online Data Analytics Workflow for Combustion Simulations

Combustion simulation-analysis workflow is composed of a primary S3D simulation and many coupled analysis components, such as iso-surface extraction [57], feature tracking [13], and volume rendering [61]. These analyses cover a broad set of algorithms that have heterogeneous data access patterns and requirements. We briefly describe the scientific background for two of the simulation analyses workflows.

S3D - Iso-surface Extraction

Extracting iso-surfaces of a varying scalar field in the computational domain is interesting and important for the S3D simulation since these iso-surfaces represent flame sheets in the turbulent flow. One challenge in extracting these iso-surface is that they are not volume filling, hence constructing them using a traditional marching cubes algorithm requires accessing the data from only a only a small portion of the entire data domain (less than 10%). However, the spatial-temporal fluctuations of turbulent flow cause the temporal iso-surface to experience "flapping." The result of this volatile behavior means that the required portion of the data domain needed for extraction may change accordingly over different time steps (i.e., it is not fixed). Furthermore, the domain scientist is often interested in the extraction of multiple iso-surfaces for different iso-values of the scalar field. In this case, multiple sub-regions of the data domain need to be accessed in the same time step.

S3D - Feature Tracking

Direct numerical simulations resolve all the relevant spatial-temporal scales and provide information on the dynamics of many interesting features, such as auto-ignition kernel, expanding or contracting flames, and extinction regions. A feature can typically identified and classified in a scalar field based on some critical points and a suitable threshold. Mostly, these features move and grow over time in the computational domain, but seldom extend to the full domain. Therefore, the feature tracking analysis can safely identify and track these features by accessing data of relevant sub-domains, under the guidance of corresponding thresholds. With the spatial-temporal changes of these features, the sizes and locations of these sub-domains change accordingly over time steps.

3.1.3 Data Staging technique - DataSpaces

DataSpaces provides distributed data interaction and coordination services to support in-situ scientific workflows on the very large scale system. It provides simple high-level abstractions that can support the dynamic and asynchronous data-intensive coupling patterns required by the targeted application workflows. It enables live data to be extracted from running simulation components, indexes this data online, and then allows it to be monitored, queried and accessed by other components and services via the space using semantically meaningful operators. Specifically, DataSpaces is built on an asynchronous, low-overhead, memory-to-memory data communication module, which allows applications to overlap interactions and data transfers with computation, and to reduce the I/O overheads by offloading data operations to staging area. This module is built upon DART [22], an open-source asynchronous communication and data transportation substrate based on RDMA.

The DataSpaces design enables it to be scalable and efficiently implemented on and across various high-performance systems and clusters. The DataSpaces implementation consists of two key components, a DataSpaces Client and a DataSpaces Server. The DataSpaces client component is integrated with the user application and provides the interfaces for interacting with DataSpaces Server. It prepares, i.e., organizes for its internal representation, and submits data queries received from the application to the DataSpaces Server. The DataSpaces server component consists of multiple server instances that run independently of user applications, on a set of nodes in the staging area. Similar to the client component, the server component provides the communication interfaces for interactions with the client components and other server component instances. The server component extends DART functionality with data storage services. Actually, It allocates in-memory storage space at each of the DataSpaces nodes and manages the memory buffers to create the abstraction of a distributed virtual storage for application data. It stores the data captured from applications locally at the DataSpaces nodes, and complements the DHT, which stores the metadata (e.g., geometric descriptor) associated with the data. The application data and the corresponding index are stored in memory to enable faster access time as compared to storing the information on disk, which thus incurs the associated latencies.

DataSpaces Workload Pattern

A data staging server which provides data indexing and transformation service, are not generally CPU-intensive. To demonstrate this workload pattern, we performed experiments using a synthetic workflow on the Titan Cray XK7 system. The data used in this workflow is based on the S3D access patterns shown in Figure 3.2. 64 synthetic simulation processes wrote a set of n-dimensional arrays to 4 staging servers with varying staged data sizes in each time step. The arrays consist of uniform double-precision floating point data.

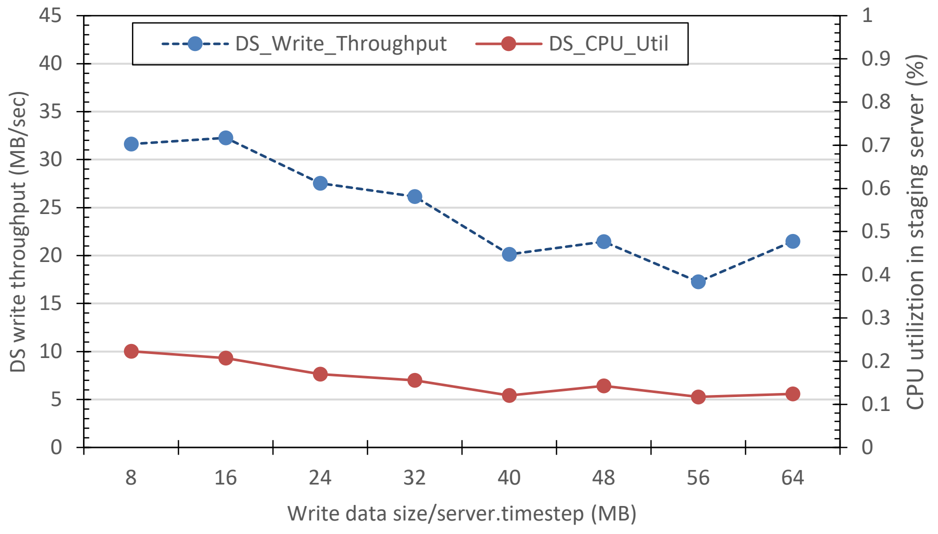


Figure 3.1: Cumulative data write throughput *(blue dot line)* and CPU Utilization = *compute time in staging/write response time (red line)* for different problem sizes when writing the entire data domain to data staging.

The amount written to each staging server per time step varied between $8M$ and $64M$

and total data size stored in the staging area through the life of the workflow varied between $320M$ and $2560M$. Using this configuration, we measured the average write throughput and CPU utilization for the data staging servers under different workloads. The results are plotted in Figure 3.1. As the size of the workload was increased, the CPU utilization of the staging cores remained consistently low, while the write throughput has already reached the peak and decreased. In this particular test, the maximum CPU utilization was observed around 22% when the write size was $8M$ for each staging server per time step. This experiment illustrates that the compute resource in the staging area is not fully utilized and the staging performance is constrained by other resources limitation such as network congestion. Therefore, there is an opportunity to leverage compute resource within staging-area to perform error detection without degrading the read and write throughput of the staging service.

3.2 Fault Tolerance Challenges

3.2.1 Data Reliability

Data loss due to failures (e.g., process failures, node failures) in any module of such complex workflows will impact the execution of the entire workflow and can invalidate the final results. Consequently, it is critical to ensure the data resilience of the end-to-end workflow. In order to address fault tolerance at extreme scales with the expected commensurate higher rates of failures and data loss [15], recent research has explored techniques for minimizing application vulnerability to failures. Various fault tolerance techniques such as checkpoint/restart, process replication [26], and erasure coding [63] have been widely studied, and have been utilized to mitigate failures in individual software components of the scientific workflow. While these approaches have been utilized to mitigate failures in individual software components of the scientific workflow, they do not address the resilience of the overall workflow. For example, in case of staging-based in-situ workflows, the underlying data staging framework supporting workflow workflows remains vulnerable to failure that lead to data loss in the case of failures.

3.2.2 Data Verification

Unlike data verification for single applications, remediation of silent errors on in-situ workflows has only recently been explored. Even worse, The impact of silent errors can potentially be amplified in the case in-situ workflows due to the dependencies, interactions and data exchanges between the components of the workflow. For example, the S3D workflow consists of S3D simulation and visualization coupling application, as illustrated in Figure 3.2. In each coupling cycle, the workflow first executes S3D simulation for several time steps, moves the data generated to the staging area, which is the processed by the analytics/visualization applications for feature extraction.

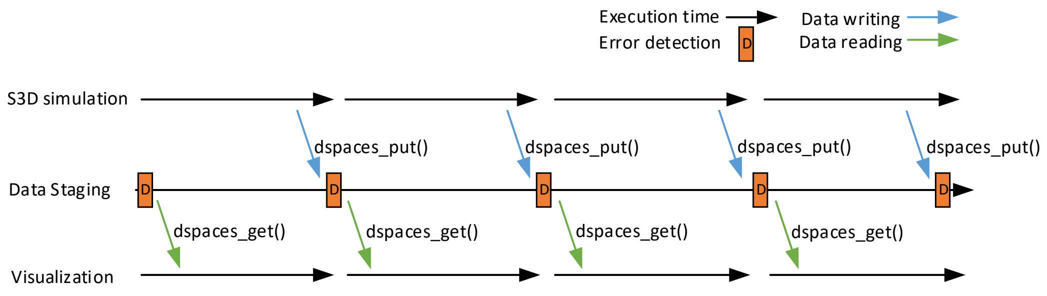


Figure 3.2: Coupling and data-exchange patterns for the in-situ S3D coupled simulation workflow.

When the workflow passes data between the S3D simulation components and the visualization components, dozens of 3D scalar and vector field components (fluid velocity, molecular species concentrations, temperature, pressure, density, etc.) are transferred using staging or other mechanisms. If there are silent errors in this data, these errors will also be propagated to the coupled components of the workflow. This will result in erroneous results in the visualization and analysis products. Even worse, in the case of multiphysics coupling, even small errors can have catastrophic results, especially to the stability of non-linear PDEs. These errors have the potential to significantly alter the accuracy of the simulation, and may not be otherwise detected by the simulation without expensive analysis on the invariant terms of the simulation.

To prevent such silent errors from propagating across components, we must identify and recover from data corruption before the data transfer is completed. While error detection algorithms can be hard-coded into applications and invoked before every data exchange, this can become quite expensive. In the case of staging-based in-situ workflows, data transfers between workflow components is done through staging, which provides an opportunity to leverage computational capabilities on the staging resources to perform error detection only on the exchanged data in an asynchronous manner. In in chapter 5, we explore the feasibility add effectiveness of such an approach.

3.2.3 Data Consistency

Recent research has also addressed fault tolerance at extreme scales and the expected higher rates of fail-stop failures. These works have explored techniques for minimizing application vulnerability to such failures. However, applying fault tolerance techniques to the application components individually does not effectually address the resiliency challenge for the whole workflows due to the data coupling at extreme scale. For example, the S3D workflow consists of S3D simulation and visualization coupling application, as illustrated in Figure 3.3. In this similar scenario that we discuss in Section 3.2.2, if the S3D simulation and analytics are protected by checkpoint/restart individually, and a failure happens in the analytics, the re-executive analytics process will get the wrong version of data from data staging considering the S3D simulation has updated the data during the analytics re-execution. (the case1 shown in Figure 3.3). Similarly, if a failure hits the S3D simulation, the re-executive simulation will unnecessarily perform the data updating operation to data staging twice, considering data has already been staged in staging in first execution (the case2 shown in Figure 3.3). These data/events inconsistency issues finally result in erroneous results of visualization and analysis products.

A further challenge to fault tolerance of entire workflows is application components in workflows exhibit different resiliency requirements, program properties and failure characteristics, which result in diversification of fault tolerance strategy among these components. For example, the stencil-based application which is commonly used as scientific simulations, such as the S3D simulation, employs local recovery strategy [31] to effectively reduce the overhead of recovery in case of frequent process failures. This recovery strategy is based on unique communication pattern of stencil-based applications which implies that multiple

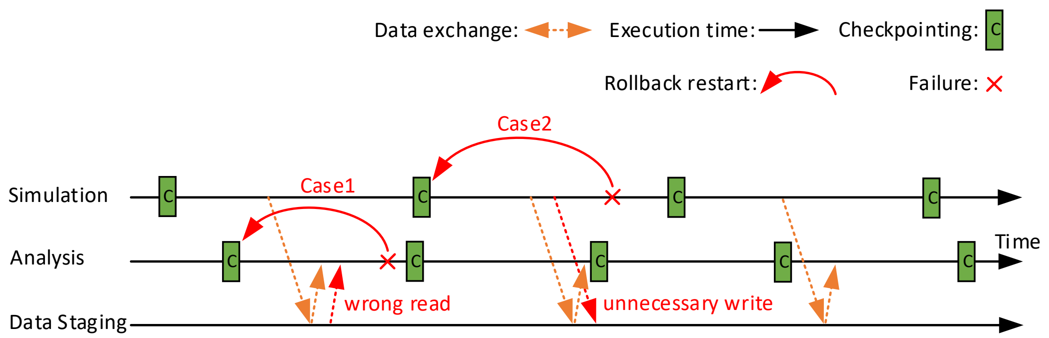


Figure 3.3: individually checkpoint/restart for the in-situ S3D coupled simulation workflow.

independent failures can be masked to effectively reduce the impact on the total time to solution. Meanwhile, as another large group of applications, the linear algebra applications typically utilize algorithm-based fault tolerance (ABFT) [10] to tolerate both fail-stop failures and silent errors. In ABFT, the application uses intricate knowledge of linear algebra to maintain supplementary, redundant data, and can be updated algorithmically to form a recovery dataset in case of failure. Although local recovery and ABFT can exhibit excellent performance and resiliency for the single application, they are less generalist approaches, and can not be appropriate for all application components in workflows. Therefore, when such types of applications with the specific fault tolerance strategies combine together into a in-situ workflow, to enable these resilience techniques working cooperatively is challenging, and constructing a workflow-level loose coupled fault tolerance framework becomes necessary. Ideally, workflow-level fault tolerance mechanisms should enable individual application components to exploit the wide area of fault tolerance techniques.

Unfortunately, naive using state-of-the-art fault tolerance approaches in workflows can not address those challenges discussed above efficiently. One possible solution to fault tolerance of in-situ workflows is to use global coordinated checkpoint/restart protocols shown as Figure 3.4. This method requires that all processes in the workflow rollback to the last valid checkpoint place, when a failure occurs. In the case of the Message Passing Interface (MPI), a very simple approach have often been taken to ensure the consistency of the snapshot: a couple of synchronizing MPI barriers can be used, before and after taking the process checkpoints, to guarantee that no application in-flight messages are present at the time of triggering the checkpoint, and thus the causal ordering of communications inside the application is avoided entirely. Global coordinated checkpoint/restart ensures a global state consistency, but it presents two concerns. As the size of the system grows, the probability of failures increases, and the minimal cost to handle such failures also increase due to frequently rollback whole workflow for recovery. Even worse, rollback of other healthy components which have no data coupling with the failed component during failures time and last checkpoint time, shown as Figure 3.4, are unnecessary and wasteful. The second is global coordinated checkpoint/restart constrain the diversity of individual application fault tolerance strategies which can efficiently reduce the resiliency cost.

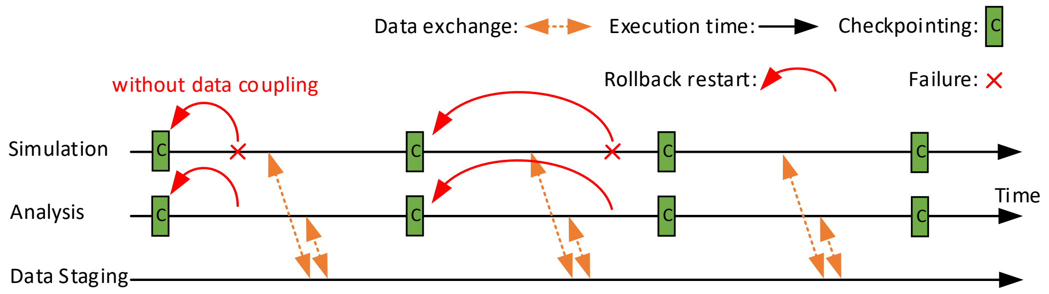


Figure 3.4: Coordinated checkpoint/restart for the entire in-situ S3D coupled simulation workflow.

Ideally, a good candidate of workflow-level fault tolerance approaches should maintain the data consistency between coupled application components efficiently, meanwhile provide a compatibility with diverse state-of-the-art fault tolerance approaches to construct a workflow-level loose coupled fault tolerance mechanism for the entire workflows.

Chapter 4

CoREC: a Scalable and Resilient In-memory Data Staging

4.1 Introduction

In this chapter, I present CoREC (Combining Replication and Erasure Coding), which is a hybrid approach to data resilience for data staging. CoREC provides the benefit of data replication, i.e., high performance, while leveraging erasure coding to reduce storage costs. CoREC uses online data classification, based on spatial/temporal locality, to determine whether to use erasure coding or replication, and balances storage efficiency with low computation overheads while maintaining desired levels of fault tolerance. Moreover, to satisfy the diverse data resiliency requirements of different workflow components, CoREC supports the use of different data redundancy schemes, such as a hybrid approach combining both replication and triplication with different erasure codes. CoREC with multilevel data redundancy (CoREC-multilevel) can dynamically decide between erasure coding and replication schemes (for e.g., duplicate, triplicate, and Reed-Solomon codes) based on the data access patterns while maintaining application-specified resiliency requirement and incurring minimal storage overhead. Furthermore, I develop optimized load balancing and conflict avoiding encoding for CoREC, as well as a low-overhead lazy-recovery scheme for the staging nodes, to alleviate overheads and interference associated with CoREC for both data-writes and data-recovery. CoREC also provides a process/node recovery solution that cooperates with the data resiliency scheme, and aims to recover failed staging servers so as to maintain the performance of the data staging framework over the lifetime of the workflow.

I have used CoREC to implement resilient data staging within DataSpaces, and have also integrated ULFM (User Level Fault Migration) [8] to efficiently recover from both processes and node failures. I have deployed the resulting resilient DataSpaces on the Titan Cray XK7 production system at Oak Ridge National Laboratory (ORNL), the Cori Cray

XC40 system at the National Energy Research Scientific Computing Center (NERSC), and the Caliburn system at Rutgers Discovery Informatics Institute (RDI2). Our experimental evaluations using synthetic workloads and the S3D combustion workflow demonstrate that CoREC maintains good storage efficiency and low latency for various use cases, supporting sustained performance and scalability even in the face of frequent node failures.

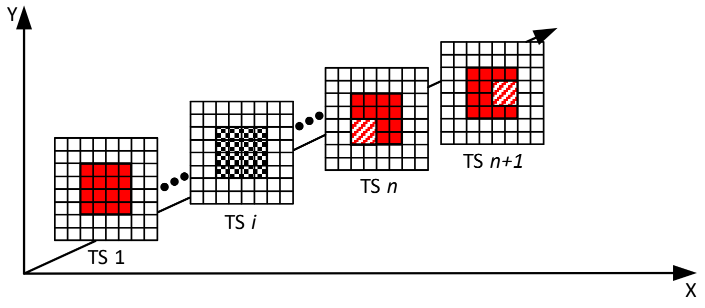
The rest of this chapter is organized as follows. Section 4.2 presents the low-latency and high-efficiency CoREC and CoREC-multilevel approach to data resilience for data staging, and Section 4.3 describes the design of CoREC. In Section 4.4, I present the implementation and evaluation of CoREC. Finally, Section 4.5 presents related work and Section 4.6 concludes the chapter.

4.2 CoREC (Combining Replication and Erasure Coding)

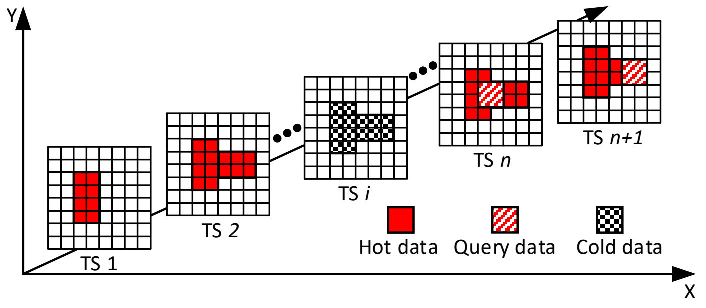
CoREC is a hybrid approach that dynamically (and intelligently) combines replication with erasure coding based on data access patterns to balance storage efficiency with computation overheads, while maintaining desired levels of fault tolerance. Specifically, CoREC uses a robust classification of data access patterns to identify *hot* and *cold* data – the key idea is to replicate the write-hot data while applying erasure coding for write-cold data. Using replication for write-hot data eliminates the expensive parity updates as I only need to update the replicas. Using erasure coding for write-cold data ensures limited object updates and dramatically reduces storage costs as compared to using a pure replication-based approach. For example, in a two-failure resiliency case, let us assume that 60% of the data is identified as write-cold, which uses erasure code ($n = 8, k = 6$), and the remaining 40% hot data objects are replicated for fault-tolerance. Here, using CoREC, I incur only 100% storage overhead compared to the 200% needed for full replication, but maintain write performance close to that of replication, assuming write-cold data are rarely updated. Note that I do not consider read access patterns in our hot/cold classification because data encoded with systematic erasure codes do not need to be decoded for reads in the absence of failures [52].

4.2.1 Classifying Data Access

CoREC utilizes the concept of write-hot and write-cold data to identify data objects as candidates for either replication or erasure coding. If a data object has been recently written/updated more than a threshold number of times within a certain interval it is considered to be hot data, otherwise it is considered to be cold data. While data access patterns in real applications can change as the application evolves, i.e., a hot data object may become cold and vice versa, access patterns in scientific applications typically exhibit high temporal and spatial data localities as the data and its access is typically defined along some discretization of a physical domain (e.g., a mesh or a grid), and the accesses are iterative in time [5].



(a) Single time step data locality case



(b) Multi time steps data locality case

Figure 4.1: An illustration of spatial and temporal data write/update patterns for a 2D data domain with $N + 1$ time steps. The solid red regions and slash regions (i.e., *hot* data) indicate data written into the staging area, while the black dot regions (i.e., *cold* data) are not updated since time step i.

During the execution of scientific simulation workflows, the simulation (e.g., S3D) issues a data write request, which writes n-dimensional data, at the end of each time-step/iteration.

Here, I use *temporal locality of objects* to indicate data objects being written/updated in consecutive time-step, and *spatial locality of objects* to refer to data objects that are near to each other in the n-dimensional space. As an illustrative example, consider a simulation that uses a 2-dimension Cartesian grid as show in Figure 4.1(a). The simulation writes data objects in region {(2, 2), (6, 6)} of the grid at time step 1, and this hot data turns cold at time step i (*temporal locality*). At time step n, another application writes/updates only a portion of that region (say region {(2,2), (3,3)}). In this case, it is very likely that the surrounding data objects in region {(2, 2), (6, 6)} (due to *spatial locality*) will also be written/updated at subsequent time steps, $n+1$, $n+2$, and $n+3$ [5].

I may go beyond this one step lookahead prediction and consider several time steps. For example, suppose that the highlighted data objects at time step 1 and step 2 are written by one application in Figure 4.1(b), and these multiple objects turn cold at time step i. If at time step n another application writes a portion of the combined regions of {(2, 2), (4, 6)} and {(4, 4), (7, 5)}, it will likely access objects in the combined region during time steps $n+1$, $n+2$, and $n+3$. This multi-time step look-ahead mechanism is beneficial because an application may have several different hot data objects at the same time-step in different regions of the grid. CoREC uses these spatial-temporal data locality attributes for multi time-step data access prediction.

While choosing candidates for replication and erasure coding, I need to consider the properties of both replication and erasure coding as described in sub-section 2.2. Since replication has advantages in terms of write performance for frequent writes but has storage overhead as compared to erasure coding, I use data access patterns to classify write-hot and write-cold data and apply replication and erasure coding techniques respectively. Specifically, newly written or updated data objects are classified as hot data. Data objects with spatial coordinates near current hot-data are anticipated to be accessed in near-future, and thus are also considered hot. The data objects with temporal locality in previous iterations/time-steps relative to the current hot data objects are also classified as hot data objects. CoREC replicates these hot data objects while all other cold-data objects are erasure coded. I use reference counters to record the access frequency of each data object. From a pool of replicated data objects, the object with the lowest access frequency is selected as a candidate for erasure coding. Once it is erasure coded, its access frequency is reset back to zero and incremented with every future access. The objects in the erasure coding pool with highest access frequencies are selected to be transitioned to replication if and only if the current storage overhead is lower than a user-specified threshold, i.e., CoREC aims to maintain storage efficiency while providing highest performance.

4.2.2 Modeling the CoREC Approach

In this section I analyze the trade-off between replication and erasure coding and the impact of data access classification on a simple hybrid approach.

If N_{level} is the data resilience level, i.e., the maximum number of simultaneous node failures that system should be able to recover from, using replication for fault tolerance requires additional N_{level} copies of each object. Therefore, the storage efficiency, which is ratio of the size of original data objects to the size of original data object plus redundant data objects, for replication is:

$$E_r = \frac{1}{N_{level} + 1}$$

Assuming that replication schemes use streaming pipelines: stream to the first node, which streams to the second node, and so on up to N. Also, the data transfer time from one server to another server is l seconds. Further assuming that each server has c second for sending the object to the remote server, the time required to guarantee N_{level} data resiliency level for one object is:

$$C_r = l \times N_{level} + c$$

Using Reed Solomon Code [47], supporting N_{level} fault tolerance with a group of N_{node} servers involves both encoding and data transfer between servers. It requires a computation overhead of $O(N_{level} \times N_{node})$ and data transfer of $N_{level} + N_{node} - 1$ data objects for N_{node} objects. Thus, the storage efficiency is:

$$E_e = \frac{N_{node}}{N_{level} + N_{node}}$$

and the time required to encode one data object is:

$$C_e = O(N_{level} \times N_{node}) + \frac{l \times (N_{level} + N_{node})}{N_{node}} + c$$

Simple Hybrid Erasure Coding

In this chapter, I use simple hybrid erasure coding to refer to a hybrid approach where candidate data objects for replication and erasure coding are selected randomly without any data classification. Suppose that an application stages n disjoint objects, and runs for a duration T while uniformly updating each object t times. Then, the resulting object update frequency is $f = \frac{T}{t}$. If the probability that an object will be replicated is P_r and the probability that an object will be erasure coded is $P_e = 1 - P_r$, then the storage efficiency for simple hybrid erasure coding (E_{hybrid}) can be computed as:

$$\frac{N_{node}}{(N_{node} \times (N_{level} + 1) \times P_r + (N_{level} + N_{node}) \times P_e)}$$

The corresponding time complexity is given by:

$$C_{hybrid} = (P_r \times C_r + P_e \times C_e) \times f \times n \tag{4.1}$$

CoREC

In CoREC I classify data objects as hot or cold based on the data update frequency f. Assuming that the object update frequency is non-uniform for hot and cold data, let these frequencies be f_h and f_c respectively, and that $f_h > f_c$. For n disjoint data objects, $P_h \times n$ hot data objects are replicated and $P_c \times n$ cold data objects are encoded in CoREC, where P_h and P_c are the percentages of hot and cold data objects in the data staging service respectively. Therefore, the time complexity for CoREC can be computed as:

$$C_{CoREC} = P_h \times C_r \times f_h \times n + P_c \times C_e \times f_c \times n \tag{4.2}$$

Since each data object in the data staging service is classified as either hot or cold, $P_c = 1 - P_h$. From equation 1, I have:

$$C_{CoREC} = (C_r \times f_h - C_e \times f_c) \times n \times P_h + C_e \times f_c \times n \tag{4.3}$$

Accordingly, the time complexity for exclusively using erasure coding $C_{erasure}$ and replication $C_{replica}$ are:

$$C_{replica} = (f_h - f_c) \times C_r \times n \times P_h + C_r \times f_c \times n \tag{4.4}$$

$$C_{erasure} = (f_h - f_c) \times C_e \times n \times P_h + C_e \times f_c \times n \tag{4.5}$$

The advantage of CoREC as compared to simple hybrid erasure coding in terms of time complexity can be computed as:

$$Gain = C_{hybrid} - C_{CoREC} = (C_e - C_r) \times P_h \times P_c \times (f_h - f_c) \times n \tag{4.6}$$

The storage efficiency for CoREC, which depends on percentage of hot and cold data (E_{CoREC}), is given by:

$$\frac{N_{node}}{(N_{node} \times (N_{level} + 1) \times P_r + (N_{level} + N_{node}) \times P_e)} \tag{4.7}$$

The prediction and classification of hot data objects depends upon the accuracy of the classifier. If the classifier is not accurate, it might classify cold data as hot data (or vice versa). Even if the accuracy of the classifier is perfect, replicating all hot data objects might be infeasible due to limited memory size. Since I can tolerate a limited storage overhead for data resiliency, in CoREC I introduce two parameters: *miss ratio* r_m and *storage efficiency constraint* S. I use miss ratio, i.e., the ratio of misclassified data objects to total hot data objects, as a measure of the accuracy of data access classification. Then, $P_h r_m n$ real hot data are classified as cold data and encoded. Thus, the time complexity for CoREC under miss ratio r_m can be computed as:

$$\begin{aligned} C_{CoREC} &= P_h(1 - r_m)C_r f_h n + P_h r_m C_e f_h n + P_c C_e f_c n = \\ &(C_r f_h - C_e f_c + (C_e - C_r) f_n r_m) n P_h + C_e f_c n \end{aligned} \tag{4.8}$$

The storage efficiency constraint S is used as an upper bound for the storage overhead that can be tolerated, which is a lower-bound for E_{hybrid} and E_{CoREC}. When $E_{CoREC} = S$, the storage efficiency constraint limit is reached and equation 7 can be solved to obtain the value of P_r as:

$$P_r = \frac{E_r \times (S - E_e)}{S \times (E_r - E_e)}$$

When $P_r < P_h$ and $P_e > P_c$, $(P_h-(1-r_m)P_r)n$ real hot data are encoded under constraint S. Thus, when CoREC hits the storage efficiency constraint, the time complexity for CoREC with miss ratio r_m can be computed as:

$$\begin{aligned} C_{CoREC} = P_r(1-r_m)C_r f_h n + (P_h - (1-r_m)P_r)C_e f_h n + P_c C_e f_c n = \\ (f_h - f_c)C_e n P_h + C_e f_c n - (C_e - C_r)(1-r_m)P_r f_h n \end{aligned} \tag{4.9}$$

Using the time complexity equations (1), (3), (4), (5), (8) and (9), I plot relative write/update cost versus the hot data percentage in Figure 4.2. When all of the data objects are cold (Marker 1 in the figure), the write performance of CoREC is the same as simple hybrid erasure coding, because data is written/updated rarely. With the increase in the hot data percentage, the time complexity for CoREC increases linearly, i.e., performance is gained due to the replication of hot data objects. If I assume that classification is accurate and there is no constraint on storage, then all hot objects are replicated and all cold objects erasure coded. In this case, the write cost will be similar to replication. When storage constraint limit S is reached (Marker 2 in the figure), some of the hot data objects will be erasure coded, irrespective of their classification, which will lead to an increase in the cost. In addition to this, if the classifier is not accurate, then there will be misclassifications, and write/update performance will be further degraded. In conclusion, between points 1 and 2 in Figure 4.2, the performance of CoREC increases due to the increase in hot data objects, but beyond point 2, the storage overhead limit is reached and objects are erasure coded irrespective of their classification, leading to a constant difference in time complexity with the full erasure coding approach, i.e., $C_{erasure}$.

Based on Equation (6) and Figure 4.2, I can deduce that CoREC's time complexity depends on the following factors: (*i*) The difference in the data access frequencies of hot and cold data objects, i.e., $f_h - f_c$. The larger the difference, the greater the benefit of CoREC. (*ii*) The difference in the time complexity of replication and erasure coding, i.e., $C_e - C_r$. The larger the difference, the greater the benefit of CoREC. (*iii*) The scale of workload n. The larger the workload, the greater the benefit of CoREC. (*iv*) The miss ratio, i.e., r_m. The lower the miss ratio, the greater the benefit of CoREC.

4.2.3 CoREC-multilevel, CoREC with multilevel data redundancy

When dealing with in-situ workflows, each application and variable potentially has different data resilience requirements. For example, large-scale, long-term simulation applications

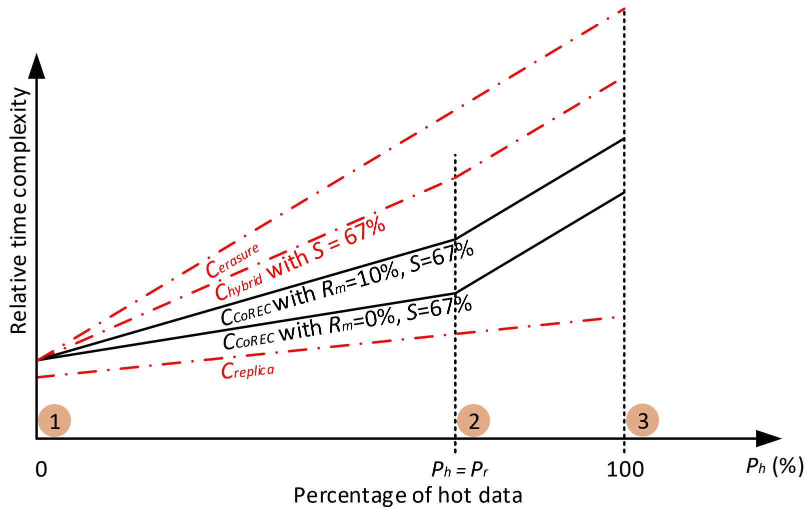


Figure 4.2: An analytic study of the relative time complexity of CoREC (C_{CoREC}) with $RS(4,3)$, and varying miss ratios(R_m) and percentages of hot data objects (P_h). The time complexity for erasure coding ($C_{erasure}$), replication ($C_{replica}$) and simple hybrid erasure coding (C_{hybrid}) is noted by red dotted lines, as baselines.

require high data resiliency due to a higher probability of failures. Meanwhile, applications in a small scale workflow may have lower data resilience requirement. Since different workflows can share the same data staging resource, using the same data resiliency scheme across the whole data staging framework is not efficient. Amplifying this concern, the level of data resilience might vary for each *variable* of an application. For example, in machine learning workflows which performing hyperparameter optimization, the hyperparameter variable is the key result of the hyperparameter tuning and failures affecting this dataset will significantly affect the entire workflow. In contrast, variables used for logging purpose are less important and unavailability/corruption of such data rarely impacts the final result of the workflow. This warrants a need to support the ability to set the data redundancy level at the granularity of variables.

In the following section, I introduce CoREC with multilevel data redundancy (CoREC-multilevel). Unlike CoREC, which only cares about data access frequency and applies a universal data redundancy for all data, CoREC-multilevel takes into account the resilience requirement of applications and variables. Specifically, I enable a varying data redundancy scheme, which corresponds to different n-way replications and erasure coding schemes based on the data resilience requirements. In CoREC-multilevel, each variables has an individual level of data redundancy, which is set by the application, and the global storage efficiency constraint is set as an upper bound of storage cost in the staging area.

Assuming that the overall cost of multilevel replications is the sum of the cost of each replication scheme (C_{ri}) weighted by the percentage of corresponding data (P_{ri}) in the total replication data, the expected cost of the multilevel method that combines n replication schemes $\widetilde{C_r}$ is:

$$\widetilde{C_r} = P_{r1}C_{r1} + P_{r2}C_{r2} + ... + P_{rn}C_{rn}$$

In the same way, the expected cost for the n erasure coding schemes $\widetilde{C_e}$ is:

$$\widetilde{C_e} = P_{e1}C_{e1} + P_{e2}C_{e2} + ... + P_{en}C_{en}$$

From these equations and equation (8), the time complexity for CoREC-multilevel C_{CoRECM} under average replication $\widetilde{C_r}$ and erasure coding $\widetilde{C_e}$ costs can be computed as:

$$C_{CoRECM} = (\widetilde{C_r}f_h - \widetilde{C_e}f_c + (\widetilde{C_e} - \widetilde{C_r})f_n r_m)nP_h + \widetilde{C_e}f_c n \tag{4.10}$$

Similarly, I can get the average storage efficiency for replication $\widetilde{E_r}$ and erasure coding $\widetilde{E_e}$ as:

$$\widetilde{E_r} = P_{r1}E_{r1} + P_{r2}E_{r2} + ... + P_{rn}E_{rn}$$

$$\widetilde{E_e} = P_{e1}E_{e1} + P_{e2}E_{e2} + ... + P_{en}E_{en}$$

The storage efficiency for CoREC-multilevel is then given by:

$$E_{CoRECM} = \frac{\widetilde{E_e}\widetilde{E_r}}{\widetilde{E_e}P_r + \widetilde{E_r}P_e} \tag{4.11}$$

4.3 CoREC System Design

CoREC is composed of three key components, i.e., the grouped replication & erasure coding based data placement scheme, the load balancing & conflict-avoid encoding workflow, and the lazy recovery strategy. In this section, I present the overall design and implementation details of CoREC, and describe these three components.

4.3.1 Data Placement

Grouped Replication & Erasure Coding Scheme

In order to tolerate concurrent staging server failures (i.e., node failure), I divide staging servers into replication groups and erasure coding groups. A replication group includes the data object and its replica, and an erasure coding group includes data objects and their parities. The grouped replication and erasure coding scheme overcomes the limitation of random replication and makes data objects able to survive concurrent failures with higher probability. Figure 4.3 shows an example of how two-way replication and erasure coding group ($k = 3, n = 4$) scheme work in a sixteen-servers data staging.

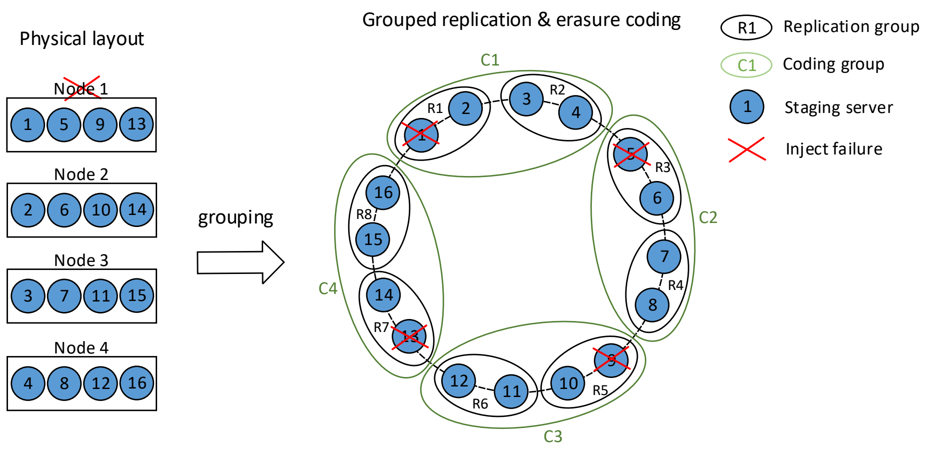


Figure 4.3: Data Objects, Replicas and Parity layout in data staging. Servers 1 and 2 are in the same replication group while servers 1, 2, 3 and 4 belong to the same coding group. This topology-aware data layout can tolerate arbitrary single node failure.

The placement of replicas and data/parity objects on staging servers in the physical organization can also have a critical effect on data resilience. In many cases, a single event such as a power failure or a physical disturbance will affect multiple devices, and greatly increases the risk of data loss. By reflecting the underlying physical organization of data staging servers, our approach can model and thereby address potential sources of correlated staging server failures. Specifically, in CoREC, I reorder the data staging server ID based on network topology and organize them in a logical ring. Each server is followed in the logical ordering by a server on a different node or cabinet so that as many as n contiguous servers belong to n different nodes or cabinets. By encoding this information into the logical network topology, our data placement policy can separate the data object, its replicas and parity objects across different failure groups while maintaining the desired distribution. As depicted in Figure 4.3, a sixteen-servers data staging which is located in 4 dedicated compute nodes can tolerate arbitrary one node failure.

4.3.2 Load Balancing & Conflict Avoid Encoding Workflow

In CoREC, data objects are encoded in staging servers during transition from replication to erasure coding. If one staging server is currently busy with a large read-write workload, assigning the encoding task to this server will impact other requests being served, as well as the encoding time. CoREC addresses this interference with a load-balancing & conflict-avoid encoding workflow. Since hot data objects are always replicated, CoREC can simply select the staging server with the lightest workload in the replication group to perform data classification and encoding operation.

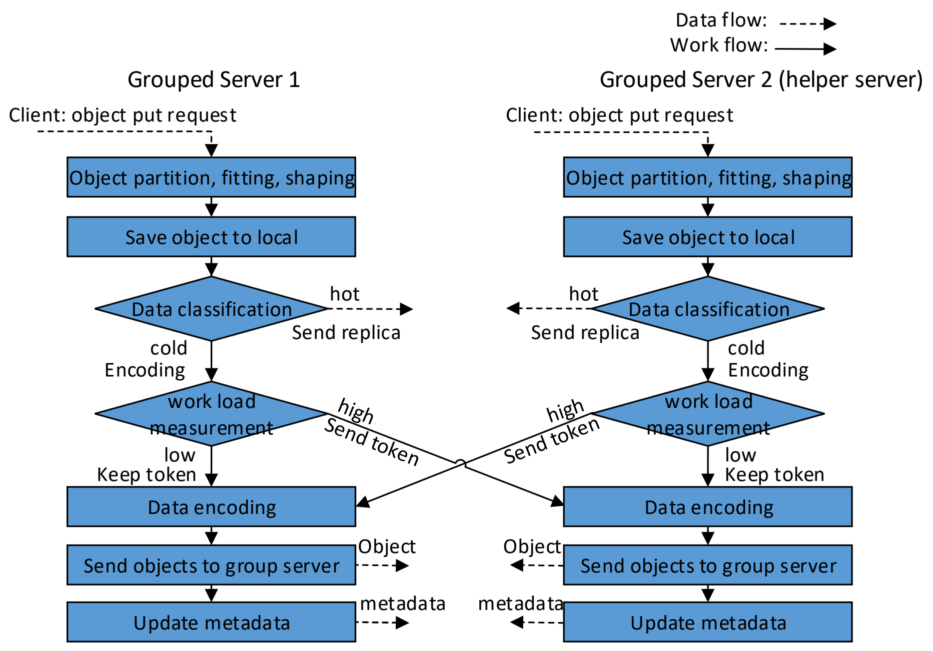


Figure 4.4: Encoding workflow in CoREC.

Figure 4.4 illustrates an encoding workflow with one server and one paired server, also called helper server, executing on a replication group of size 2. The encoding workflow is triggered by the server when it receives an object-put request from a client. Once server receives and pre-processes the data object, data classification component classifies data objects and make decision for the data resilience approach based on data frequency and storage efficiency constraint. After that, the workload measurement component decides whether to encode locally or let the helper server encode based on its workload level. If local node's workload is high, then it sends the replica node (node with replica data) an encoding token to perform erasure coding. Otherwise, the server performs encoding locally. After the server performs the encoding operation, it sends data & parity objects to other servers in the erasure coding group.

The encoding workflow comprises four principal components. First, a data fitting and partition component pre-processes the data objects into a specific size and shape. Second, a data classification and encoding component classifies data object and makes it resilient. Third, a workload measurement component measures a server's workload level based on the frequency of client read-write requests. Finally, a data/parity object consistency mechanism provides atomic encoding processing for each data objects. In a replication group, all servers share one encoding token and the server can get the encoding token only if it has a low workload. Only the server that holds an encoding token can perform an encoding operation, which ensures that exactly one stripe is placed in the coding grouped servers. It also ensures that the less busy server in the group performs more encoding operation than the busier one and workload is balanced throughout the coding group.

4.3.3 Data Size & Geometric Shape

While very small data objects suffer from metadata overheads, larger data objects have relatively smaller metadata overheads and achieve better throughput during asynchronous communication such as RDMA [23]. However, large-sized data objects increase the processing time required for data encoding, decoding, replication and transportation [60]. This leads to longer data access latencies. Thus, an appropriate object size is required to balance metadata overhead and data access latency.

Algorithm 1 Geometric partitioning and fitting of an object

```
Require: Data Object ($object$), metadata, dimension ($n$), fitting size ($size$);
Ensure: Fitting data objects ($object[m]$), metadata ($metadata[m]$);
  $N \Leftarrow 1$
  $object[m] \Leftarrow object$
  while $N \neq 0$ do
    if $\exists$ $obj$ in $object[m] > size$ then
      get maximum boundary size of $obj$ in dimension $n$
      partition boundary to half
      partition $obj$ to half
      $metadata[m] \Leftarrow metadata$
      $object[m] \Leftarrow obj$
    else {Object is fitting}
      return $object[m]$, $metadata[m]$
    end if
  end while
```

In order to fit data objects into desirable size and shape on the servers, the data fitting and partition component in CoREC uses Algorithm 1. In this algorithm, I first set a range of target data object sizes. When a staging server receives a data object that is larger than the range, I partition the object into halves along the longest geometric dimension. This is done repeatedly until all sub-objects fall into the range of target size. This simple binary partition algorithm ensures that data objects do not exceed a threshold size. Partitioning in this way ensures a balance between the size of objects and the quantity of objects. Under perfect conditions, every object can be partitioned into regular and uniform n-dimensional objects.

4.3.4 Recovering Data Staging Server Failures

Existing large-scale resilient storage solutions typically use an aggressive data recovery strategy[18]. Whenever a failure on one or more servers is detected, all lost objects are recovered and re-generated onto active servers immediately. The problem with such an aggressive data recovery scheme is that it requires significant resources to recover data from a failure. Decoding operations and data transportation may consume considerable network and computing resources in a short time window. These overheads eventually hinder the application read-write requests. In CoREC, I propose a new lazy recovery scheme with a time limit on delayed data recovery. As shown in Figure 4.5, recovering data staging servers from failures involves four key steps: (*i*) failure detection, (*ii*) data recovery in the degraded mode, (*iii*) process recovery, and (*iv*) data recovery in the lazy recovery mode. CoREC introduces two data recovery modes (the degraded mode and lazy recovery mode) for data recovery.

Failure Detection

In CoREC, the function of detection and handling of failures is delegated to ULFM-enabled MPI. As a proposed extension of the MPI standard, ULFM [8][9] includes mechanisms to tolerate fail-stop failures without the need to restart all processes linked to the MPI communicator. I leverage ULFM to tolerate and recover from such failures in the data staging area. ULFM guarantees that MPI operations involving communication should return an

ERR_PROC_FAILED error code if the runtime detects a process failure in the data staging communication area. ULFM-specific return codes are captured using MPI's profiling interface and no changes in the MPI runtime itself are required. In data staging frameworks, the data exchange between applications happen via reads/writes from/to the staging area. The reads/writes are facilitated via asynchronous RDMA. When staging server processes fail or are unavailable, RDMA error codes can also used to detect these failures.

Degraded Mode

Once a data staging server detects a process failure, it distributes failure notifications to the remaining data staging servers. The data staging area is then shrunk to remove the failed staging servers and the rest of the staging servers switch to a degraded mode, as shown in Figure 4.5. In this mode, only the requested data is re-constructed, and transferred to the client. This temporarily re-constructed data is discarded once the read request is served. The reconstruction of failed data objects in the read-path increases read latency. Experimental evaluation results for the reading performance in degraded mode are presented in Section 4.4.

Process Recovery

To be able to recover from process failures, CoREC reserves a few staging server processes as backup processes. The number of backup processes is determined by the choice of erasure code and server process density on a node. For example, if CoREC is initialized to uses erasure code with $n = 8, k = 6$, the goal is to tolerate 2 node failures per 6 nodes. If the each node has 8 server processes, then for process recovery, CoREC initiates 16 backup processes per 48 staging processes. When failures are detected, these idle data staging processes will be activated and merged with the old data staging processes group. These newly activated processes will be reassigned the same rank numbers as failed ones. An alternative approach is to spawn new processes instead of use processes from a previously prepared process pool, if this is supported by the job scheduler.

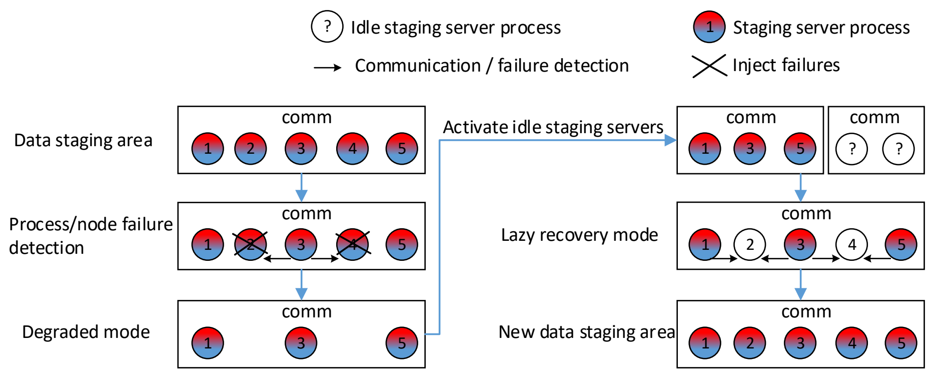

Idle staging server process
Staging server process
Communication / failure detection
Inject failures
Data staging area
comm
Process/node failure detection
Degraded mode
Activate idle staging servers
Lazy recovery mode
New data staging area

Figure 4.5: Data and process recovery in data staging area.

Lazy Recovery Mode

After a replacement server joins data staging, CoREC switches to the lazy recovery mode. In this mode, each object on the failed server will be recovered immediately after it is queried or updated. The recovery of all other remaining objects are triggered based on the time-limit set for delayed data recovery. The time-limit setting depends on the fault tolerance requirement for data objects and the overall MTBF of the system. Normally, too long of a time-limit constraint results in an unacceptably high risk of permanently losing the data as it increases the chance of multiple failures in the same group. On the other hand, too short time-limit constraint risks interfering with the application's regular requests in the same way as aggressive recovery. Specifically, CoREC uses $\frac{1}{4}MTBF$ as the recovery timeline constraint. In many data-intensive simulation applications, most of the failed objects will be recovered much earlier than the end of the timeline due to high-frequency of update and query requests.

4.4 Experimental Evaluation

This section describes the implementation details of CoREC and presents an experimental evaluation using synthetic benchmarks as well as the S3D combustion simulation and analysis workflow [16].

CoREC is implemented on the top of DataSpaces [23], an open-source data staging framework. The schematic overview of the runtime system is presented in Figure 4.6. In addition to modifying several existing components of DataSpaces for the integration, the system architecture introduces three key new components in data resiliency module: Local Object Management, Object Transportation, and System Status Monitor. The Local Object Management component maintains local data objects, replicas, parity objects, and metadata. It also stores the data object classification information in addition to performing the encoding, decoding and object preprocessing tasks. I use the Jerasure open-source library [50] to perform encode/decode operations. While evaluation results demonstrate the efficacy of CoREC when using Reed-Solomon code, the Jerasure library offers a variety of erasure codes to choose from and it is straightforward to change the erasure code used in CoREC. The Object Transportation component synchronizes data objects, replicas, parities, and metadata while managing the transportation of objects between different staging servers. Server's workload monitoring and failure detection is performed by the System Status Monitor component. In order to recover from failed staging server processes, I also introduce an additional process resiliency module. This module manages a spare process pool and implements the detection and handling of staging server failures using ULFM, which offers a set of fault tolerance mechanisms for MPI applications.

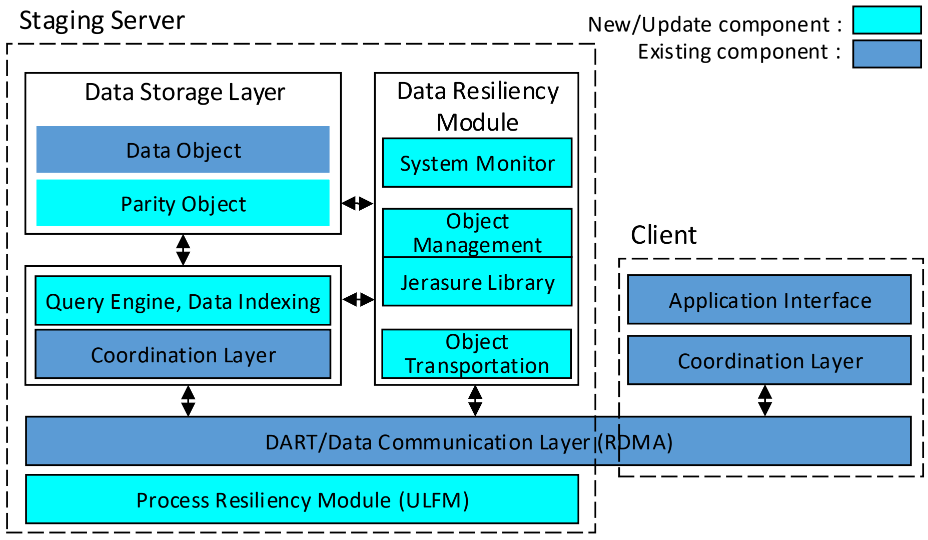
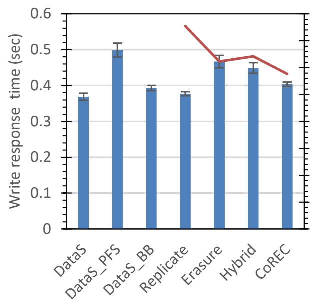


Figure 4.6: System Architecture

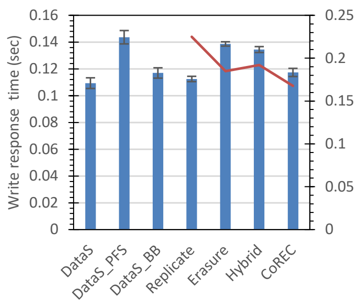
Synthetic Experiments

Our synthetic experiments were performed on both, the ORNL Titan Cray XK7 system and the NERSC Cori Cray XC40 system. These experiments evaluate the read and write performance of applications with different data read and write patterns, when they use CoREC for resilient data staging. To better understand the performance and effectiveness of our approach, I selected five test cases with common data reading and writing patterns used by real scientific simulation workflows. In these cases, I assume that scientific applications write data to a 3-dimensional global space (*data domain*). I also assume that data is written in multiple iterations (*time-steps*) as described in five test cases below. I compared CoREC with five other fault tolerance mechanisms: DataS_PFS (replicates all data objects and places replicas on the parallel file system), DataS_BB (replicates all data objects and places replicas on burst buffer nodes), Replication (replicates all data objects and places replicas on peer staging servers), Erasure Coding (encodes all data objects locally and places data/parity objects on peer staging servers), and Hybrid Erasure Coding (data objects are classified and selected for replication/erasure coding under the LRU algorithm and a defined constraint on storage overhead). I additionally compared our results to the performance of data staging without any fault tolerance. In order to evaluate the balance between the write response time and the storage cost for various data resilience technique, I introduce *write efficiency*, which is a ratio of application's observed write response time to the storage efficiency of the data resiliency technique. The low write efficiency value indicates a better balance between time and storage cost for data resilience. The setup of these experiments is described in Table 4.1. The experimental results are presented in Figure 4.7 (collected on Cori) and Figure 4.8 (collected on Titan), along with a detailed discussion and analysis of these results.

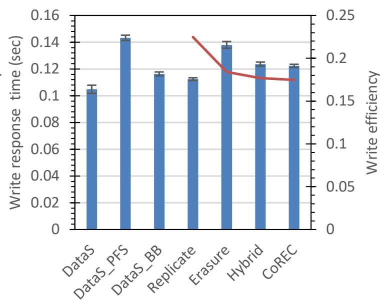
1) **Case 1** - *Write the entire data domain in each time step*: In this case, the data of the entire domain is written at every simulation time step. Since there is no data replication, encoding, data movement and metadata synchronization overhead, the data staging without fault tolerance has the best relative data write response time. For the fault tolerance approaches, although the replication approach with unlimited storage constraint on peer memory or burst buffer does not incur the overhead of data encoding, I/O operation and



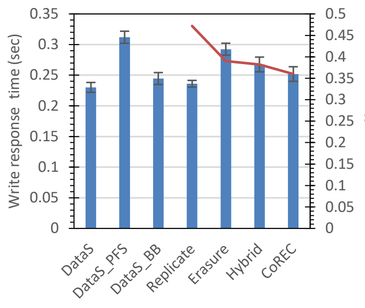
(a) Case 1



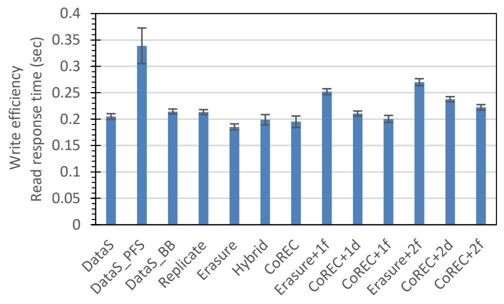
(b) Case 2



(c) Case 3



(d) Case 4



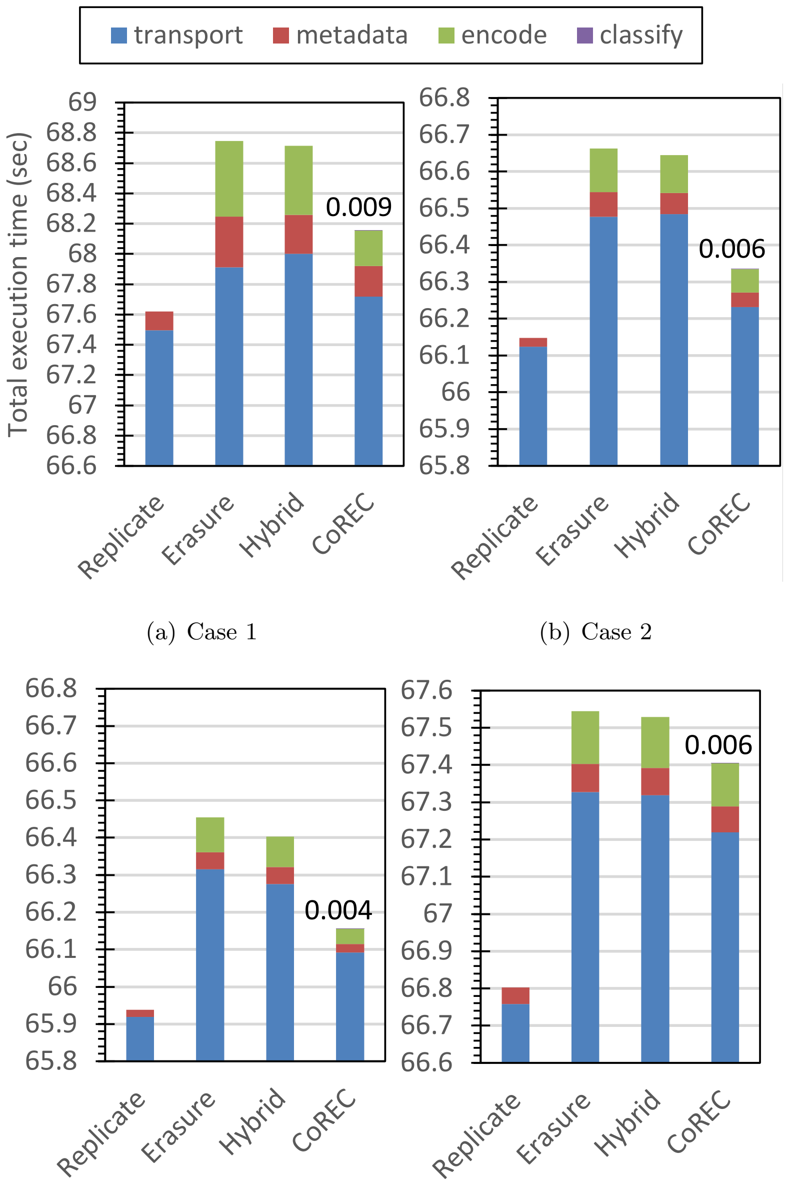
(e) Case 5

Figure 4.7: Average data write and read response time *(blue bars)* and Write Efficiency = *Write response time/Storage Efficiency (red line)* of different data resilience mechanisms for the five test cases using different writing patterns. *DataS:* Data staging without fault tolerance; *DataS_PFS:* Data is stored in PFS for resilience; *DataS_BB:* Data is stored in Burst Buffer for resilience; *Replicate:* Data is replicated in peer memory for resilience; *Erasure:* Data is erasure coded for resilience; *Hybrid:* hybrid erasure coding with LRU data classification; *CoREC+1d and CoREC+2d:* CoREC in degraded mode with 1 and 2 server failures; *CoREC+1f and CoREC+2f:* CoREC in lazy recovery mode with 1 and 2 server failures; *Erasure+1f and Erasure+2f:* Erasure coded data staging with an aggressive recovery strategy under 1 and 2 server failures.

Total number of cores	$64 + 32 + 8 = 104$
No. of parallel writer cores	$4 \times 4 \times 4 = 64$
No. of staging cores	8
No. of parallel reader cores	32
Volume size	$256 \times 256 \times 256$
In-staging data size (20 TSs)	2560MB
No. of replica	1
No. of data objects	3
No. of parity objects	1
Coding technique	Reed-Solomon Code
Storage efficiency for hybrid erasure coding	67%
Storage efficiency lower-bound for CoREC	67%

Table 4.1: Experimental setup for synthetic tests.

extra data transportation overhead, these approach have only achieved 6.9% and 2.6% smaller write response time in comparison to CoREC respectively. Meanwhile, storing the replicated data into PFS gets the worst write access performance and the longest total workflow execution time due to intensive I/O operations. Also, the result for the erasure coding method shows the second worst performance for both write access and total workflow execution time because of the overhead associated with frequently encoding the original data objects and the placement of data/parity objects on peer servers. Although only a portion of data objects are erasure coded in hybrid erasure coding, frequently switching between replication and erasure coding approach on the same data object makes this approach's write performance just slightly better than the erasure coding approach and has longest total transportation time. For CoREC, due to the write-intensive workload, the workload balance and conflict-avoid encoding workflow plays a vital role in minimizing the interference to regular request. CoREC gets achieves a decrease of 48.7% and 53.2% in encoding time and an improvement of 13.5% and 10.1% in the write response time, relative to erasure coding and hybrid erasure coding. The lower-bound constraint for storage efficiency in CoREC causes some data objects to be erasure coded, even if they are hot, and this leads to a 6.9% increase in write-time as compared to replication.



(a) Case 1 (b) Case 2

(c) Case 3 (d) Case 4

Figure 4.8: Breakdown of the total execution time (in seconds) for the workflows in Figure 4.7. *transport:* Time spent in data movement; *metadata:* Time spent to update the distributed metadata; *encode:* Time spent to perform data encoding; *classify:* Time spent for data classification in CoREC (listed as number).

2) **Case 2** - *Write the entire data domain in multiple time steps*: In this case, the entire data domain is divided into 4 subdomains, and each subdomain is written in a time step. This means that in every 4 time steps, the entire data domain is written. Since each

subdomain has the same write access frequency, all data objects in that subdomain are either hot or cold. However, CoREC leverages its multi-time step look ahead mechanism to efficiently convert data objects from cold to hot i.e., moving from erasure coding to replication. Thus, CoREC has better (around 12.7%) performance improvement for write response time and 38.4% decrease in the encoding time with respect to hybrid erasure coding, while incuring an overhead of 4.3% in the write response time over replication. In addition, the conflict avoid encoding workflow and fewer data conversion from replication to erasure coding contributes to having smaller data transportation overhead than hybrid erasure coding.

3) **Case 3** - *Write a subset of the data domain at a higher frequency than others*: In this case, data objects of a subdomain in a particular domain is written at higher frequency and data objects in other subdomains are written just once. This setup addresses the presence of hot spots in the data domain. Both CoREC and hybrid erasure coding with LRU can easily identify these hot data objects and apply the corresponding resiliency technique. Since erasure coding always select all data objects as candidates for erasure coding, CoREC improves the write response time by 11.3% and 1.1% and decreases encoding time by 50.4% and 16.5% respectively, while increasing the write response time by just 8.8% as compared to replication.

4) **Case 4** - *Write subsets of the data domain with random access pattern*: This case differs from the previous case as the subdomains of the data domain are randomly chosen for writing/updating. The random access pattern reduces the accuracy of the data classifier, which is based on temporal and spatial locality. However, the workload balance and conflict-avoid encoding workflow optimizations enhance the performance of CoREC by 13.8% and 5.8% and decrease encoding time by 14.6% and 17.8% compared to erasure coding and hybrid erasure coding respectively.

Figure 4.8 shows the breakdown of the normalized execution time for workflows in aforementioned cases in failure free case. The plots show that CoREC has lower overheads compared to hybrid erasure coding and pure erasure coding in all cases. CoREC has less data transport time than erasure coding and hybrid technique because fewer erasure coded objects incur updates and it minimizes the parity update operations, which leads to less encoding time also. While replication has better performance, it should be noted that it suffers from high storage overhead.

These experiments demonstrate although CoREC involves extra cost for data classification and switching resilience approaches between replication and erasure coding, it can still get the more benefit from dynamically identifying hot/cold data and performing corresponding data resilience approaches. Therefore, CoREC achieves the overall better performance and lower cost than full erasure coding and hybrid erasure coding with LRU approaches in four synthetic cases.

5) **Case 5** - *Read entire data domain in each time step*: The data of the entire domain is read for every time step. In this case, the replication method, either in peer memory or burst buffer, has a similar read response time to data staging without fault tolerance, meanwhile replicating data to the PFS results in the worst performance among all fault tolerance approaches. Since, erasure coding splits original data objects into small objects and distributes them among the staging servers, a single read request can be distributed across multiple servers and consequently erasure coding, hybrid erasure coding, and CoREC have better read response times than both replication and the original data staging technique. I also performed experiments for various cases of reads as we did for writes, but the results are not presented in this section due to the similar patterns as case 5. I also evaluated read response time of CoREC in the presence of failures. In degraded mode, the read response time increases by 7.93% and 21.7% for single and double server failures respectively, as compared to failure-free case. However, when using lazy recovery, the read response time increases by 2.66% for single failure and 13.9% for double server failures as compared to failure-free case. While I demonstrated that CoREC performs better on average than both erasure coding and hybrid erasure coding, replication might seem like a good choice for fault tolerance. However, I also need to consider the storage overhead associated with each fault tolerance mechanism. I also plot the ratio of write-response time and storage efficiency in Figure 4.7. It can be seen that, data staging without fault-tolerance provides best performance along with best storage efficiency. On the other hand, fault-tolerance introduces overheads on both write response time and storage efficiency. Among the fault-tolerant mechanisms, CoREC provides the best balance for storage efficiency and write response

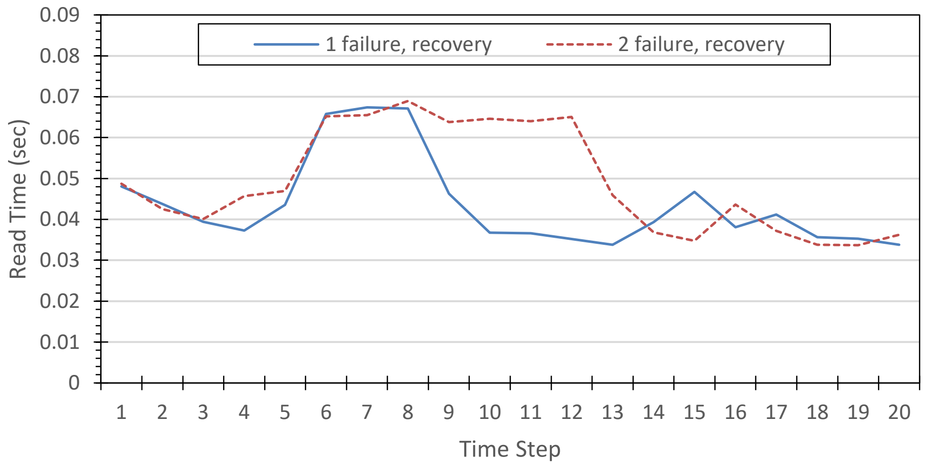


Figure 4.9: The average read response time for reading the entire data domain with 1 and 2 failures, along with failure recovery, for 20 time steps. The first failure occurs at time step 4, and second failure occurs at time step 6. First failure-recovery begins at the 8^{th} time step and another recovery is initiated at the 12^{th} time step, and they end at time steps 9 and 13 respectively.

time in all the data access patterns studied.

In order to study the impact of lazy recovery in CoREC, I also perform the experiment on Titan and plot the read response time at every time step for 20 time steps in Figure 4.9. For a single failure case, I inject a staging server failure at time step 4 and recover it at time step 8. For the two failure case, I inject a first staging server failure at time step 4 and a second failure at time step 6, and then start recovering them at time step 8 and 12 respectively. In both the cases, the entire data domain was read for all time steps. I observe that, unlike aggressive recovery, our lazy recovery approach does not trigger data recovery for the failed server immediately, which may result in an increased data read response time. From time step 8 to time step 9, our approach gradually recovers unavailable data objects, which leads to a nominal increase in the data read response time for recovery from multi-server failure. After time step 14, the data read response time resets back to what it had been before the failure was injected.

No. of cores	4480	8960	17920
No. of simulation cores	$16 \times 16 \times 16 = 4096$	$32 \times 16 \times 16 = 8192$	$32 \times 32 \times 16 = 16896$
No. of staging cores	256	512	1024
No. of analysis cores	128	256	512
Volume size	$1024 \times 1024 \times 1024$	$2048 \times 1024 \times 1024$	$2048 \times 2048 \times 1024$
Data size (GB)	160	320	640
No. of replica	1	1	1
No. of data objects	3	3	3
No. of parity objects	1	1	1
Storage efficiency	67%	67%	67%

Table 4.2: Configuration of core-allocations, data sizes, and data resilience for the three test scenarios on 4480, 8960 and 17920 cores.

Large Scale S3D Experiment

I also performed large-scale tests for CoREC using the lifted hydrogen combustion simulation workflow using S3D[16] and an analysis application on Titan, and compared it to pure replication and erasure codes. CoREC was tested using three different core count (4480, 8960 and 17920) and corresponding grid domain sizes so that each core was assigned a spatial sub-domain of size $64 \times 64 \times 64$. In terms of the data access pattern, in this experiment, the S3D simulation wrote the vector field component pressure of entire domain to data staging at each time, which was processed for feature extraction by the visualization application later. For comparison purpose, I also ran S3D without data staging, S3D with data staging but without resilience, and S3D with data staging and resilience. The cumulative time for reading/writing data over 20 time steps was measured. The core configurations, the data region assignments, and data resilience for our experimental setup are summarized in Table 4.2.

Figure 4.10 and Figure 4.11 illustrate the experimental results for the S3D coupled simulation and analysis application workflow, for various resiliency settings. Since the PFS (parallel filesystem) based S3D does not have data staging and the data is saved to disk, it has the longest read and write response time. While data staging without resilience shows best performance, it is not able to recover from failures. Among the resilient data staging techniques studied, CoREC reduces the write response time by 7.3%, 14.8%, and 5.4% as compared to pure erasure coding on 4480, 8960, and 17920 cores respectively. In comparison

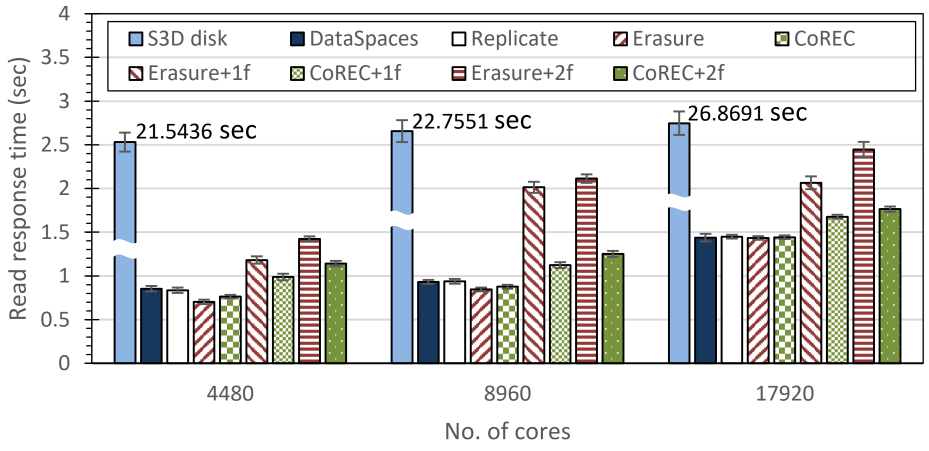


Figure 4.10: Comparison of the cumulative data read response time using the S3D and coupled analysis workflow on Titan.

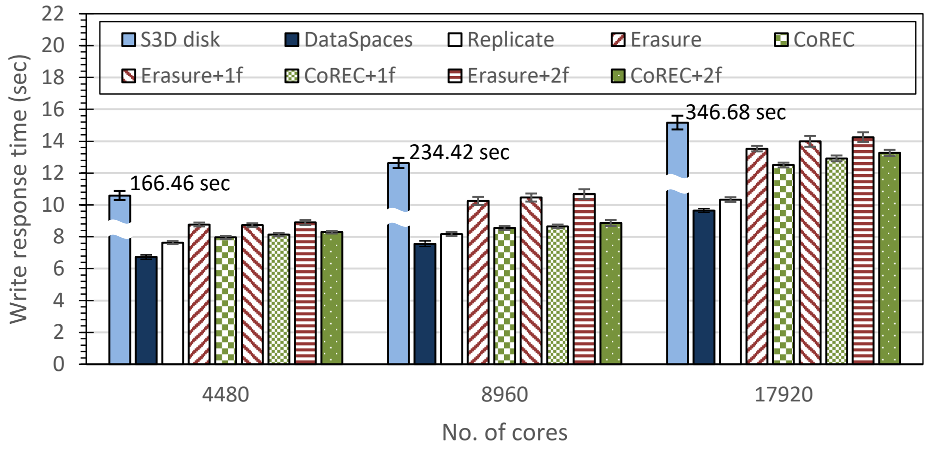


Figure 4.11: Comparison of the cumulative data write response time using the S3D and coupled analysis workflow on Titan.

to replication, CoREC has an overhead of 4.2%, 5.3%, and 17.2% in write response time on 4480, 8960, and 17920 cores respectively. It can also be seen that in the presence of failures, CoREC reduces the read response time by up to 40.8% and 37.4% for one and two server failures respectively as compared to pure erasure coding.

These results show that CoREC demonstrates good overall scalability, better storage efficiency with small overheads for different processor counts and data sizes, while providing data resiliency for extreme-scale HPC systems.

4.4.1 Experiments with Node Failures

The goal of these experiments is to demonstrate that CoREC is capable of handling data and process recovery under high-frequency node failures.

Experimental Setup

I have deployed CoREC with ULFM on the Caliburn cluster which consists of 560 compute nodes, each containing two 18-core Intel Xeon E5-2695v4 processors, 256 GB of main memory and an Intel Omni-Path Host-Fabric interface adapter at Rutgers Discovery Informatics Institute (RDI2). In this experiment, I evaluate the overhead related to data recovery for staging node failures. In our experiments, a staging node failure is equivalent to N staging server processes failures at same time, where N is the total number of processes per dedicated staging node. In order to perform these experiments, node failures are injected by simultaneously sending SIGKILL signals to all the staging server processes running on a particular node. Since the data objects, parity objects, replicas and metadata are stored in process memory, killing server processes makes such data unavailable. This is consistent with the behavior of real node failures. In our experiments, one compute node of the Caliburn cluster runs 8 staging processes, which translates to $N = 8$ for node failures. Unless specified otherwise, all tests have been repeated 5 times. I ran some preliminary experiments for high values of MTBFs (such as 5 minutes, 10 minutes or 30 minutes, etc.) and observed negligible recovery overheads relative to the total execution time. Subsequent experiments were ran under MTBFs less than a minute. The data size for each data staging server was 50 MB. I also ran this workflow with a failure-free case as the baseline. The setup of these experiments is described in Table 4.3.

Total number of cores	$1024 + 256 + 128 = 1408$
No. of parallel writer cores	$8 \times 8 \times 16 = 1024$
No. of staging cores	256 (32 nodes)
No. of parallel reader cores	128
Volume size	$128 \times 128 \times 256$
In-staging data size (50 TSs)	3.2GB
No. of replica	1
No. of data objects	3
No. of parity objects	1
Coding technique	Reed-Solomon Code
Storage efficiency lower-bound	67%

Table 4.3: Experimental setup for node failures tests.

Experiment Description and Results

For node failure experiments, I studied the read/write response time for different data/process recovery strategies in the data staging. I also evaluated the total overhead for workflows in order to empirically demonstrate the low cost of data recovery and small latency impact. I performed these experiments on a 256 core (32 node) data staging area. Figure 4.12 and 4.13 plot the average read/write response time for different frequencies of node failures injected for a synthetic workflow with a total execution time of about 150 seconds. The synthetic failure rates range from 18 to 150 seconds, and the corresponding total number of failures range from 8 processes (1 node) to 64 processes (8 nodes), as noted on top of each bar, over a total time period of about 150 seconds.

Figure 4.12 shows the cumulative read-response time for 50 time steps. The read-response time increases with the failure rate. In the worst case (8 node failures), the read-response time increases 9.58% in degraded mode, and 6.77% in lazy mode as compared to the failure free case (FF). For the write-response time shown in Figure 4.13 shows the similar trend with the read-response time. The write-response time increases by only 2.41%

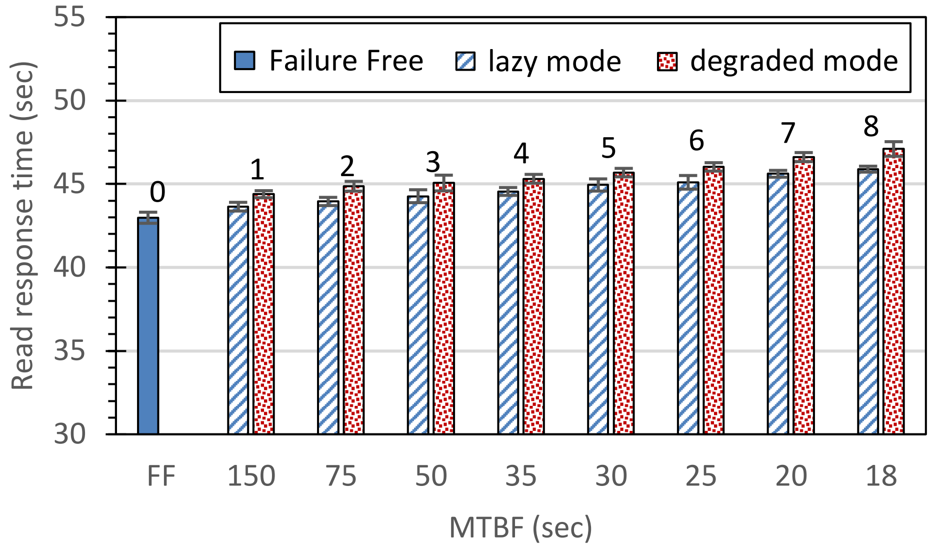


Figure 4.12: Comparison of the cumulative data read response time using the synthetic workflow on Caliburn. *FF:* in the x-axis represents CoREC in failure free case.

in degraded mode, and 1.13% in the lazy mode for an MTBF of 150 seconds. For the MTBF of 18 seconds, the write-response time increases 10.88% in degraded mode, and 8.11% in lazy mode as compared to FF. As a whole, the experiment results demonstrate that grouped replication and coding scheme with lazy recovery mode can tolerate frequent node failures with minimal overhead.

I also study the impact of data and process recovery on the total workflow execution time, which includes the overhead of data replication, encoding, decoding and process recovery. Specifically, I compare the end-to-end workflow execution time under varying MTBFs and the failure-free case. The leftmost bar in Figure 4.14 shows the total workflow execution time for the failure-free case. By using lazy recovery I obtain much lower performance penalties even at the higher failure rates. In comparison to the baseline failure-free case, the total execution time is increased by only about 0.8% in degraded mode and by 0.5% in lazy recovery mode for the MTBF of 150 seconds. For the case of failures occurring every 18 seconds, the workflow execution time increases by about 5.69% in degraded mode and by 4.25% in lazy recovery mode. In short, CoREC can handle data recovery under frequent node failures, with total overheads of up to 5.69% in degraded mode, 4.25% in lazy mode for the worst case scenario.

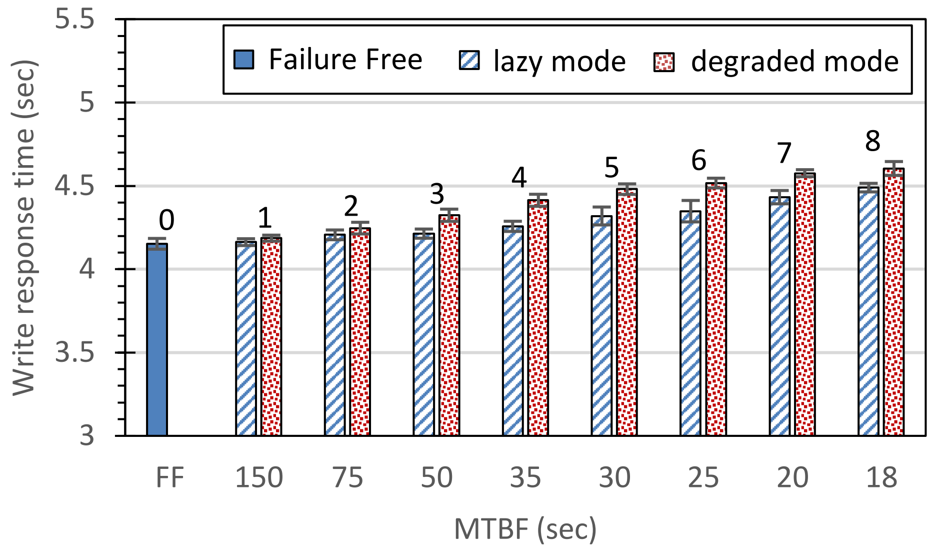


Figure 4.13: Comparison of the cumulative data write response time using the synthetic workflow on Caliburn. *FF:* CoREC with failure free case.

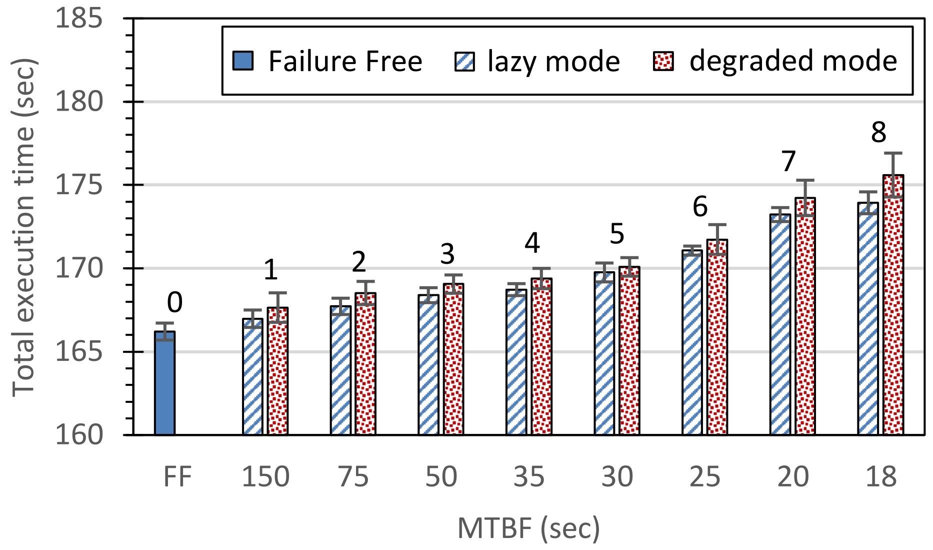


Figure 4.14: Comparison of the total execution time using the synthetic workflow on Caliburn. *FF:* CoREC in failure free case.

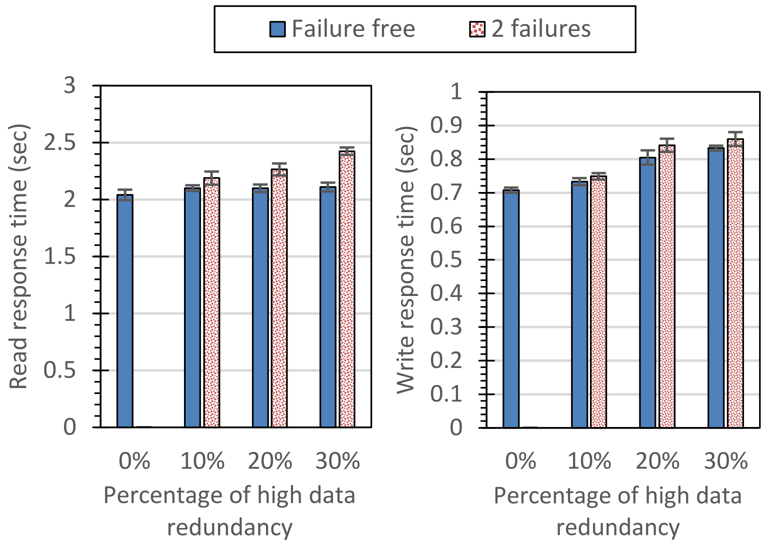
4.4.2 Experiments for CoREC with multilevel data redundancy

In this section, I demonstrate that CoREC-multilevel can efficiently provide varying levels of data redundancy based on application requirements. Although CoREC-multilevel uses different redundancy schemes to satisfy resiliency requirements, the storage efficiency constraint is maintained by managing the trade-off between storage and performance, and duplication/triplications and erasure coding. Since CoREC-multilevel is intelligent enough to make dynamic decisions based on data access characteristics, I did not observe a significant impact on application performance. The synthetic workflow writes two variables into data staging. These two variables are intended to representative high and low redundancy datasets. Within the workflow, the coupled components write and read these two variables into the data staging area concurrently in each time step, and the response time is measured.

Total number of cores	$512 + 120 + 512 = 1144$
No. of parallel writer cores	$8 \times 8 \times 8 = 512$
No. of staging cores	120
No. of parallel reader cores	512
Volume size	$256 \times 256 \times 256$
In-staging data size (20 TSs)	3200MB
Replication for high data redundancy	Triplication
Replication for low data redundancy	Duplication
Coding for high data redundancy	RS(6, 4)
Coding for low data redundancy	RS(6, 5)
Storage efficiency lower-bound	66.7%

Table 4.4: Experimental setup for multilevel redundancy tests.

For high data redundancy, I use $RS(6,4)$ and triplication to tolerance concurrent failures in two arbitrary staging servers. For low data redundancy, I use $RS(6,5)$ and duplication to tolerate one staging server failure. To study the impact of lazy recovery in the presence of concurrent failures for CoREC-multilevel, I insert failures into two staging servers concurrently. During a total-runtime of 20 time-steps the failures were inserted only once, and then recovered in subsequent time-steps using the lazy recovery mechanism. Since 2 staging servers have failed, only high-redundancy data can be recovered. As CoREC-multilevel can re-direct write requests to other staging servers when failures are detected, the low-redundancy data sets for all time-steps are available except for the particular time-step when the failures were introduced and detected. In our experiments, the total data exchanged between readers and writers is kept constant. I vary the percentage of high-redundancy data with regards to total data from 0 to 30, while keeping a constant storage efficiency constraint in CoREC-multilevel. Since the total data is comprised of high and low-redundancy data, it can also be viewed as varying low-redundancy data from 100% to 70%. Figure 4.15 shows both cumulative read-response and write-response time. A failure free case is considered as baseline in our tests. The details of the experimental setup is listed in Table 4.4



(a) Read response time of variables (b) Write response time of variables

Figure 4.15: Read and write response time of variables under the percentage of high data redundancy in CoREC-multilevel. *Failure free:* CoREC-multilevel in failure free case.

In Figure 4.15(b). I observed that when the percentage of high-redundancy data increases, the cumulative write-response time increases in Figure 4.15(b). This increase corresponds to the higher complexity or overheads of erasure coding and replication for higher data redundancy. When the amount of high-redundancy increases, more data can be recovered and correspondingly the write response is affected to a higher degree in presence of failures. When all of the data is low redundancy, the data cannot be recovered when multiple failures are inserted. Thus, no write and read-response time is reported for 0% high-redundancy data. As compared to the failure-free case, CoREC-multilevel has an increase in cumulative write response time by around 2.2%, 4.5% and 3.2% as compared to failure free case.

When no failures are present, the read-response time remains fairly constant across varying percentages of high and low dataset. In the presence of failures, read-response times are directly impacted because data needs to be recovered first and then sent to the reader application. Under the presence of concurrent failures and data recovery, the read response time of CoREC-multilevel increases by around 4.1%, 7.9% and 15.5% as compared to the failure free cases, when high-redundancy data is set to 10%, 20%, and 30% respectively.

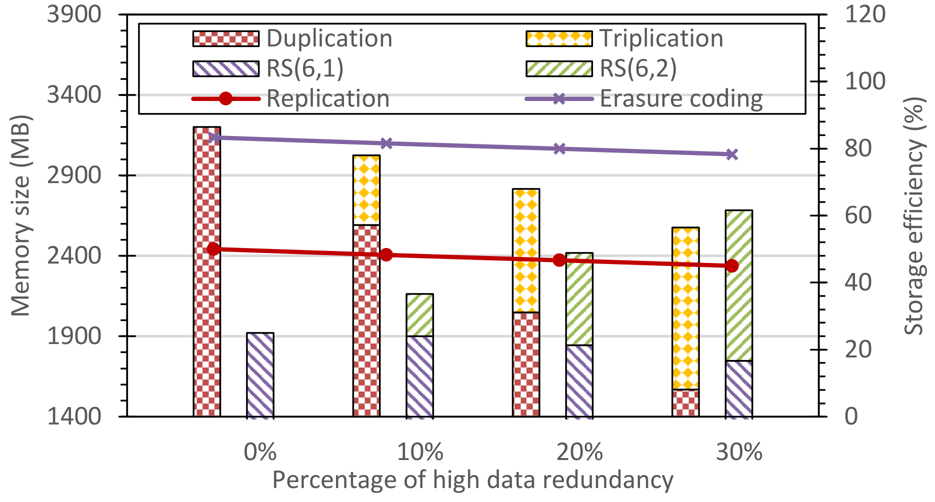


Figure 4.16: Storage cost and efficiency for the percentage of high data redundancy in test scenarios.

To understand the impact of different data resiliency techniques on the overall storage capacity of the staging area, I measure the total memory consumed by the data in the staging area and also show the storage efficiency in Figure 4.16. When the percentage of high redundancy data increases, the storage efficiency of replication decreases from 50% to 45%. Similarly, the storage efficiency of erasure coding also decreases from 83.3% to 78.3%.

Correspondingly, the memory consumption for replication decreases from $3200MB$ to the least $2576MB$, and the consumption for erasure coding increases from $1920MB$ to the largest $2683MB$ so as to maintain total storage usage is constant. Since the total amount of raw data written to the staging area is fixed, the overall storage efficiency also remains the same for different percentages of high and low redundancy data. From these results, I can infer that CoREC-multilevel is performed with light overhead and can provide various data resiliency levels based on application needs.

4.5 Related Work

Supporting resilience in contexts other than in-situ/in-transit data analytics, such as Checkpointing [34],[1],[58],[32] and Replication/Erasure Coding [18],[59] has been widely studied, but there are limited research efforts focussed on in-situ/in-transit data processing systems. The study in [40] exploits the reduction style processing pattern in analytics applications and reduces the complications of keeping checkpoints of the simulation and the analytics consistent. Research efforts in [41] use a synchronous two-phase commit transactions protocol to tolerate failures in high performance and distributed computing system. In comparison to these efforts, our data resilience approach specifically targets data staging based in-situ workflows, and is more flexible, asynchronous and scalable. Furthermore, it can handle dynamic execution and failure patterns across multiple applications that are part of in-situ/in-transit workflows.

Burst buffers are being increasingly used in HPC systems [39],[55],[38],[2] with the initial goal of relieving the bandwidth burden on the parallel file systems by providing an extra layer of low-latency storage between compute and storage resources. CoREC can easily be extended to use burst buffers. In this setting, CoREC would store the hot data in local DRAM memory and keep the cold data in the non-volatile storage layer on the burst buffers nodes to achieve faster read/write performance for workflows generating large amount of data. However, burst buffers themselves are not immune to failures. Furthermore, due to the difference in the physical typologies in burst buffers provided by vendors, the fault model and resiliency for burst buffers are complicated and remain an open challenge. The fault tolerance of burst buffers needs to explored further before any of the data resiliency methods, explored in the paper, can be used on such devices.

Recent research [36],[43],[4],[35],[44] indicates temporal and spatial properties of failures in different software and hardware components. CoREC can easily tolerate such concurrent and correlated process/node failures using topology aware grouped replication and erasure coding schemes discussed in Section 4.3.1. I believe CoREC can also be adapted to tolerate other classes of failures such as a GPU failure, which will potentially result in data loss at staging servers, by extending the failure detection and handler mechanism to those failures.

While aspects of CoREC may appear conceptually similar to Cocytus [63], where replication is used for small-sized and scattered data (e.g., metadata and key) and erasure coding is used for large data (e.g., value), CoREC uses data access frequency rather than data size for data classification. In contrast to Cocytus, which is designed for cluster storage system, CoREC targets in-situ/in-transit data processing on large-scale HPC systems.

4.6 Summary

This chapter addresses data resiliency for staging-based in-situ/in-transit workflows, and presented the design, implementation and evaluation of CoREC and CoREC-multilevel, a scalable hybrid approach to data resilience for data staging frameworks that used online data access classification to effectively combines replication and erasure codes, and to balance computation and storage overheads. Then, this chapter introduces an implementation and deployment of CoREC on top of the DataSpaces on the Titan Cray XK7 at OLCF, Cori Cray XC40 at NERSC, and the Caliburn system at RDI2. Finally, this chapter evaluated its effectiveness and performance by using both synthetic benchmarks and real world large scale S3D application on large-scale HPC cluster, and demonstrated that CoREC can dynamically classify data objects based on data-driven access pattern and provide efficient data recovery in the presence of frequent process and node failures. The source code for our prototype implementation of CoREC is publicly available at **https://github.com/shaohuaduan/datastaging-fault-tolerance**.

Chapter 5
Staging Based Silent Error Detection Framework

5.1 Overview

A silent error is an unintentional change to a bit in memory. These undetected bit flips impact the correctness and performance of applications and workflows [15]. These types of error events are already present and impactful in high-end computing. Although various error-detection techniques for silent errors, such as ABFT [10] and time-series predictions [7], have been widely studied, these studies have generally been in the context of single applications rather than workflows, which are a composition of multiple interacting component applications. As extreme scale workflows are often long-running and the final result of the workflow dependends upon intermediate results, a silent error or data corruption in any component can invalidate the entire execution of the workflow, with substantial impact. As a result, it is important to detect and isolate silent errors in a component application as early as possible and to contain the propagation of these errors between components. However, simulation and analysis applications in workflows are commonly CPU-bound, so frequently performing error detection in such applications will significantly degrade the overall performance of the workflow. In chapter 3, we observed that staging resources tend to be relatively less heavily loaded compared to the computations nodes that run the workflow components. Given this resource-usage pattern, this chapter presents a staging based silent error detection framework to offload error detection task to data staging, which enables efficient error detection without suffering performance degradation.

The rest of the chapter is organized as follows. In Section 5.2, we introduce our staging-based error detection framework. In Section 5.3, we evaluate our approach using various synthetic and real-world scientific simulation workflows. Section 5.4 provides details of various related work, and Section 5.5 summarizes this chapter.

5.2 Error Detection in Staging

In this section, we first model our staging-based error detection framework. We then analyze it by simulating scientific workflow events and present factors that influence the rate of error detection in staging. Finally, we propose optimizations for implementing error detection approaches in the data staging area and leveraging GPU resource (if available) to ameliorate the impact of performing error detection on common I/O operations.

5.2.1 Modeling Error Detection in Staging

Notations: Let C, T_c, and T_s denote the cost of checkpointing, the optimal time cycle for checkpointing, and the time cycle of error detection in staging, respectively. Let R_s represent the *recall* of error detection in staging, which is the proportion of detected errors over all errors that have occurred in the course of execution. If D is the cost of error detection in a given checkpoint component and D_s is the cost of error detection in staging, we assume that $D_s << D$, since the amount of data exchanged via the staging area is much smaller than checkpoint data size. W_{base} denotes the base execution time of a workflow without any resilience technique. The notations that are used in the model are listed in Table 5.1.

Fault tolerance model: Scientific workflows typically employ a checkpoint/restart mechanism to periodically perform checkpointing tasks to enable rollback to the previous state in case of a fail-stop failure. To deal with both fail-stop failures and silent errors, checkpointing tasks are coupled with error detection (or verification) mechanisms. To guarantee that checkpoint data is error-free, error detection is performed just before checkpointing tasks [6]. If the verification fails, it can be assumed that a silent error has occurred and the application must rollback to the previous checkpoint for error-free data.

Figure 5.1 shows a typical workflow with a checkpointing scheme combined with error detection. In each checkpoint cycle, coupled simulations *simu*1, *simu*2 can access staging servers and fetch or write data from/to the servers. Since data is stored in the staging area, we can improve the fault tolerance scheme by leveraging idle compute resource in the staging area to perform data verification. In this way, during each checkpoint cycle, there

Symbols	Explanation
C	Cost of checkpointing
T_c	Optimum time cycle for checkpoint
D	Error detection cost in the checkpointing component
T_s	Time cycle for error detection in data staging
D_s	Error detection cost in data staging
T_r	Time interval from last checkpoint to the error detection in data staging
R_s	*recall* of error detection in data staging
P_s	*precision* of error detection in data staging
P_f	Probability of silent errors during execution time
T_w	Time cycle for workflows pattern
W_{base}	Workflow execution time without resilience techniques and failures

Table 5.1: Symbols summary

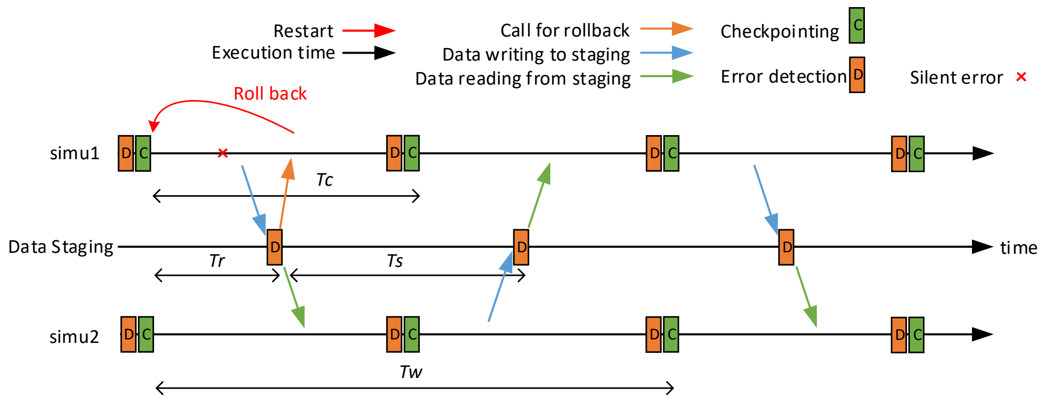


Figure 5.1: An illustration of a typical workflow with two coupled simulations *simu*1, *simu*2. Simulations alternate in exchanging data via data staging. For fault tolerance, each simulation performs checkpointing and silent error detection tasks based on their optimal checkpoint time cycles.

will be multiple error detections in the staging for early detection of silent errors and to save re-execution time. For the checkpointing scheme, a workflow makes a global checkpoint and performs error detection based on its optimal checkpoint time cycle for fail-stop failures. In the following analysis, we compare the approach where error detection occurs only at checkpoint-time with the approach that performs error detection in both staging and at the checkpoint. The goal of performing error detection in staging is to minimize the expected execution time of the workflow, $E(W)$.

Error detection in checkpoint component

Suppose that the probability of a silent error happening in the workflows is P_f. Let T_{DC} be the maximum cost for error detection and checkpoint of *simu*1, *simu*2: $Max(D_1 + C_1, D_2 + C_2)$. Let T_{cC} denote the maximum time chunk for two consecutive checkpoints: $Max(T_{c1} - C_1, T_{c2} - C_2)$. Also, error detection in the checkpoint component can always fully verify data. Then, the expected execution time of the workflow ($E(W_c)$) can be computed as:

$$E(W_c) = \frac{T_w}{T_c}(T_{DC} + P_f \times T_{cC}) + W_{base} \tag{5.1}$$

Error detection in checkpoint and staging

We now add an extra error detection in the staging area. Firstly, let us suppose that any silent error happening during T_s can be detected. That means *recall* and *precision* for error detection in staging are $R_s = 100\%, P_s = 100\%$. In this scenario, the workflow executes for an extra D_s time until error detection in staging completes and sends an error notification back. Then, the workflow needs to rollback to the previous error-free checkpoint and discard $T_r + D_s$ time worth of work. Therefore, the expected execution time of the workflow ($E(W_s)$) can be computed as:

$$E(W_s) = \frac{T_w}{T_c}(T_{DC} + P_f(T_r + D_s)) + W_{base} \tag{5.2}$$

Then, we can make our model more realistic by only assuming that the *precision* of error detection in staging $P_s = 100$. In this case, the staging can detect R_s percent of silent errors and the remaining $1 - R_s$ percentage will be detected by the checkpoint component. Thus, the expected execution time of the workflow $(E(W_s))$ can be updated to:

$$\frac{T_w}{T_c}(T_{DC} + P_f((1 - R_s)T_{cC} + R_s(T_r + D_s))) + W_{base}$$

When the *recall* rate is R_s and the *precision* is P_s for the error detection in staging, the staging will generate $1 - P_s$ percentage needless rollback operations due to the introduction of false positives in detection. The penalty for false positives is $T_p = (1 - P_s)(T_r + D_s)$. Thus, the expected execution time of the workflow $(E(W_s))$ can be updated to:

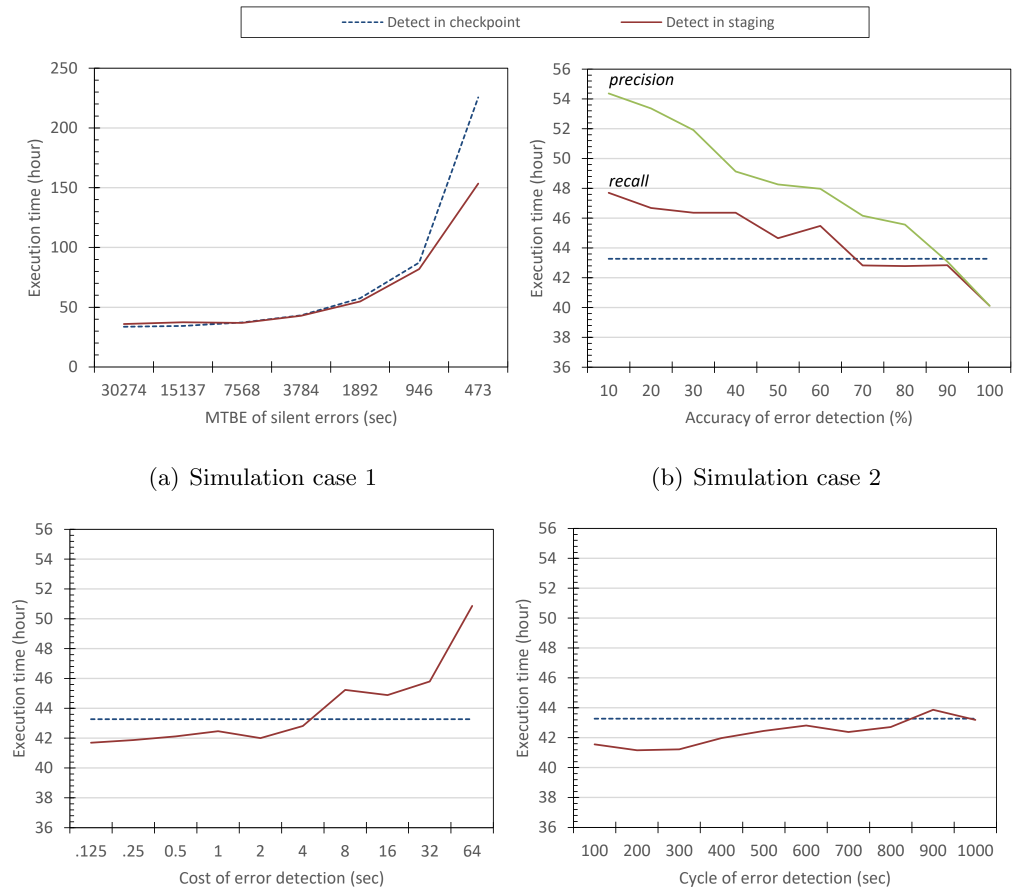
$$\frac{T_w}{T_c}(T_{DC} + P_f((1 - R_s)T_{cC} + R_s(T_r + D_s) + T_p)) + W_{base}$$

From the above equations, it can be expected that adding error detection in the staging area can reduce the total workflow execution time in the presence of errors, which can be computed as:

$$E(W_c) - E(W_s) = \frac{T_w}{T_c}P_f(R_s(T_{cC} - T_r - D_s) - T_p) \tag{5.3}$$

5.2.2 Simulation and Analysis

In order to determine the expected advantage of this approach in practice, we simulate the execution of a scientific workflow using the parameters (i.e., $MTBF$, C, and T_c) suggested in [6], which are also listed in Table 5.2. For $MTBF$ and $MTBE$, we calculate the $MTBF$ and $MTBE$ of one computing node, which is 4.3 years for fail-stop failures and 7.9 years for silent errors. The overall $MTBF$ and $MTBE$ are obtained by dividing the per-node $MTBF$ and $MTBE$ by the total number of nodes running the workflow. For example, when 65536 nodes are used in extreme scale systems, the overall $MTBF$ and $MTBE$ are $2064s$ for fail-stop failures and $3784s$ for silent errors. The silent errors are generated by following an exponential distribution of parameter $\lambda = \frac{1}{MTBE}$. The disk checkpoint cost C is close to $300s$, which is typical of many state-of-the-art platforms, such as Hera [6]. Under the assumed $MTBF$ for fail-stop failures and the checkpoint cost C, we get the optimum checkpoint cycle of $1491s$ through Daly's formula [54]. Furthermore, we assume that there is no relation between $MTBF$ and $MTBE$, and $MTBF$ will be kept constant in the simulation test cases. For error detection, we assume that the error detection in a checkpoint has a cost of $100s$ with the *recall* and *precision* of 100% and that error detection in staging has a cost of $4s$ with a *recall* 70% due to error detection being performed on a subset of the global data domain. We assume that the computation and communication cost of the workflow is $81920s$ ($22.6hr$), which does not include checkpointing and error detection tasks.



(a) Simulation case 1

(b) Simulation case 2

(c) Simulation case 3

(d) Simulation case 4

Figure 5.2: The simulated total work flow execution time comparison between error detection in checkpoint component (*blue dot line*) and error detection in both checkpoint and staging (*solid line*).

Table 5.2 describes the inputs of the emulation program. The emulation results are presented in Figure 5.2, followed by a detailed discussion and analysis of each. The reported results are the average of 100 experimental runs.

In simulation case 1, where we vary the workflow's $MTBE$, our approach has better performance than performing error detection only in checkpoint as long as $MTBE$ is under

No. of case	**Case** 1	**Case** 2	**Case** 3	**Case** 4
$MTBE$	/	$3784s$	$3784s$	$3784s$
C	$300s$	$300s$	$300s$	$300s$
D	$100s$	$100s$	$100s$	$100s$
T_c	$1491s$	$1491s$	$1491s$	$1491s$
P_s	100%	/	100%	100%
R_s	70%	/	70%	70%
D_s	$4s$	$4s$	/	$4s$
T_s	$600s$	$600s$	$600s$	/
T_w	$81920s$	$81920s$	$81920s$	$81920s$

Table 5.2: The parameter configuration for workflow time sequence emulation.

$3784s$. We performed analysis for different values of *recall* under *precision* = 100% and different values of *precision* under *recall* = 100% in simulation case 2. It can be seen that when *recall* is greater than 70% or *precision* is greater than 89%, error detection should be performed in the staging area. For the cost of error detection when it is less that $4s$, error detection in staging outperforms the detection in checkpointing, which is illustrated in simulation case 3. To analyze the impact of T_s, we also varied T_s in simulation case 4 and observed that when $T_s < 800s$, our approach is beneficial and it maintains similar performance for higher values.

Based on Equation 5.3 and the analysis above, we can deduce that the advantage of enabling error detection in the staging depends on the following factors: (i) The frequency of silent errors $MTBE$ (higher will see more advantage from the in-staging approach); (ii) The *recall* and *precision* of error detection in staging R_s, P_s (higher is better); (iii) The cost of error detection in staging D_s (lower is better); (iv) The frequency of staging operation T_s (performance will improve with less frequent staging operations).

In addition, unlike *recall*, lower *precision* can significantly deteriorate the performance of error detection in staging. Part of the reason for this is that lower *precision* indicates high false positive, which can unnecessarily rollback the workflow frequently. In contrast, lower *recall* corresponds to high false negative, which means that CPU cycles are wasted performing error detection that is unable to detect silent errors. Generally, the cost of error detection in staging is much less than the cost of re-executing the workflow.

5.2.3 Implementing Error Detection in Staging

While the integration of error detection within a data staging framework seems straightforward, naïve application of error detection technique in the data staging can result in significant performance degradation and low error detection accuracy. One of the challenges for error detection in staging is that the data staging framework stores data temporarily during data exchanges, so an entire spatial dataset is not always available. Another challenge for error detection in staging is that data staging has relatively limited compute resource, so an error detection approach having a long computation time could significantly degrade the performance of staging servers. Thus, an ideal error detection approach should maintain high accuracy, especially high *precision*, in error detection. Also, the approach should have light overhead and minimal impact on common data staging operations, such as data $put()$ and $get()$.

In the following section, we introduce two optimization techniques which can achieve the twin goals of lightweight execution and high accuracy. We use Spatial Local Outlier Measure (SLOM) [53] as an example to illustrate how to integrate an error detection approach in the data staging efficiently. SLOM is a type of spatial outlier detection approach that can capture both spatial auto-correlation (non-independence) and spatial heteroscedasticity (non-constant variance), which are common features of scientific spatial datasets. In SLOM, the effects of spatial auto-correlation are factored out by a measure $d(o)$. The variance of a neighborhood is captured by $\beta(o)$, which quantifies the oscillation and instability of an area around o.

$$SLOM(o) = d(o) * \beta(o)$$

CPU-GPU Hybrid Staging

In an extreme-scale workflow, staging servers need to process thousands of requests. Thus, any delay caused by error-detection will interfere with servicing other requests, amplifying the impact, and potentially causing a significant delay across the whole workflow. To minimize this impact, we propose offloading workload to GPUs, when available. GPUs are now widely used as general purpose devices in HPCs. They enable significant speed-ups in performance for scientific computation and analysis tasks. Since silent error detection approaches are computation intensive tasks, we can leverage these GPUs to achieve significant acceleration of error detection. Specifically, CPU cores in the dedicated staging node can be used to serve communication and regular staging operations, such as indexing and data storage, while the data from local CPU memory is offloaded to GPUs and GPU kernel performs the spatial outlier detection.

Algorithm 2 Error Detection Processing

```
Input: spatial data $P$, meta data $Q$
Output: outlier value $O_{max}$, error decision $flag$
 1: dspaces_put($P$)
 2: $flag = 0$
 3: Detector_Kernel($Q$, $P$)
 4: if $O_{max} > threshold$ then
 5:     $flag = 1$
 6: end if
 7: insert $P$ into staging
 8: update meta data $Q$
 9: Detector_Kernel($Q$, $P$)
10: for each point $p$ in $P$ do
11:     $D = d(p)$
12:     $B = beta(p)$
13:     $O = D * B$
14:     $O_{max} = Max(O, O_{max})$
15: end for
```

There are two main challenges in processing large amounts of data on the local GPU device: relatively limited bandwidth and small memory size. We address these issues as follows. The data staging process first partitions the larger geometric dataset into several smaller tiles, such that each tile fits into the GPU global memory. Next, the tiles are stored in a queue, which is located in CPU memory and is ready to be offloaded to GPU global memory. While the GPU threads process the tile, the next tile in the queue is prefetched into GPU global memory to hide the data transfer latency. This CPU-GPU hybrid staging approach can also be leveraged by CPU-only application workflows, such as the S3D simulation workflow. The pseudo code of spatial outlier error detection in CPU-GPU hybrid staging is provided in Algorithm 2. This can be easily extended to other error detection methods.

Tuning precision and recall in Staging

Although *recall* and *precision* both play a very important role for error detection in staging, from section 5.2.2, we can deduce that compared with *recall*, a lower *precision* can significantly deteriorate the performance of error detection in staging. In addition, many outlier-based error detection approaches output a score quantifying the level of 'outlierness' of each data point, but this value does not provide a concise summary of the small number of data points that should be considered as silent errors. Specifically, in SLOM simply identifying the data point with the maximum SLOM value as a silent error will virtually always produce a false positive.

In the following, we introduce dataset/feedback training which can tune *precision* and *recall* for outlier based error detection approaches so as to minimize false positives with acceptable false negatives. During the dataset training, we inserted synthetic bit flip errors into the staged dataset and monitored the minimum value that can just cover all synthetic errors, then we mark it as the error detection *threshold*. For simplification, in our implementation the synthetic silent errors are uniformly distributed along the dataset and bit position. The *threshold* may be different for different simulation time steps, datasets, and staging servers.

During the feedback training, at the initialization of runtime, staging servers systematically calculate a SLOM value for each data point and keep track of the maximum observed SLOM value, as error detection threshold *threshold*. The staging servers then start to perform error detection for the available dataset. When the staging server finds one silent error candidate and the workflow is re-executed from the last checkpoint, it goes into feedback training mode. The staging server compares the new SLOM value with the previous one which was detected as an error. If the new SLOM value in that area is under the *threshold*,

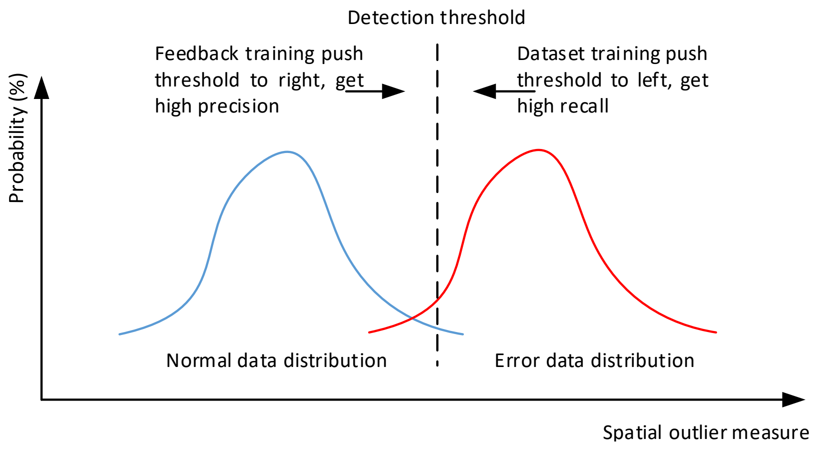

Detection threshold
Feedback training push threshold to right, get high precision
Dataset training push threshold to left, get high recall
Probability (%)
Normal data distribution
Error data distribution
Spatial outlier measure

Figure 5.3: An illustration of the behavior of and relation between dataset training and feedback training.

that means the staging server accurately identified the silent error. Otherwise, the previously detected silent error is a false positive and *threshold* is raised. Figure 5.3 illustrates that feedback training tunes the detection *threshold* towards the error data distribution and achieves higher *precision* of error detection. Similarly, dataset training pushes the detection *threshold* toward a normal data distribution and achieves higher *recall*.

5.3 Experimental Evaluation

In this section, we evaluate staging based error detection framework, implemented as an extension of DataSpaces [21], an open-source data staging framework, and perform evaluations using both real and synthetic applications in the presence of silent errors.

In Section 5.2, we described our model for error detection in staging. In this section, we provide a specific implementation for the model on Titan HPC system. Figure 5.4 shows the overall system configuration for error detection in the workflow. In the Titan Cray XK7 system, each compute node contains a 16-core 2.2GHz AMD processor with 32 GB of DDR RAM, a NVIDIA Kepler accelerator (GPU) with 6 GB of GDDR RAM and a Cray Gemini high-speed interconnect. Although subsequent experiments are only run on Titan, recently deployed machines such as Summit (a 4,608-node supercomputer at Oak Ridge National Laboratory (ORNL)) also have GPUs attached to each nodes and our approach can benefit from these GPUs. Please note that our implementation of SLOM using CPU-GPU is just

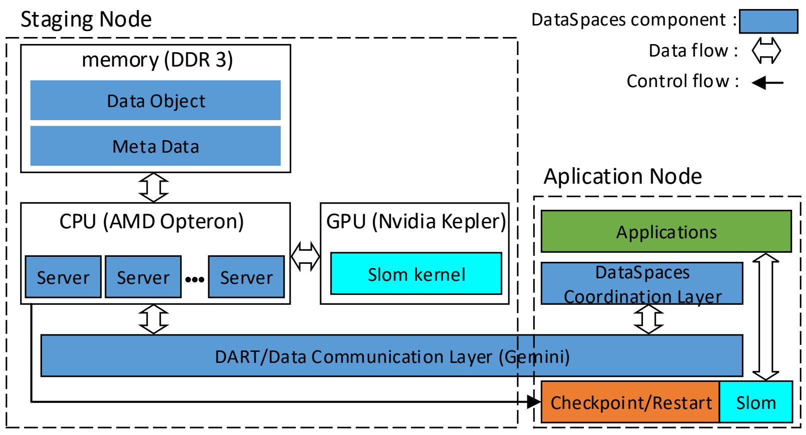


Figure 5.4: Implementation of error detection in CPU-GPU hybrid staging

an example of an error detection method. The goal of the paper is to leverage idle staging resources to perform error detection using the error detection method of choice for staging based in-situ scientific workflows. The system architecture permits a key component: dedicated staging nodes, distinct from the application nodes. Simulation and analysis applications' processes run on application nodes. Before an application process checkpoints its entire data into the parallel file system, the process also performs an error detection locally to ensure data correctness. The DataSpaces coordination layer in application nodes sends dataset to staging servers through the DART layer (DataSpaces' data communication layer). After a staging server stores the data in local memory, it performs error detection in the staging nodes. For CPU-GPU hybrid staging, it will offload error detection into local GPU if the GPU compute resource is available for the dedicated staging node. When a silent error is detected, the staging server sends error message to application's checkpoint/restart components and triggers a process restart. In order to do that, we update the DataSpaces interface function *ds_put()* so that it can return the detection result back to applications and indicate if there is a silent error in the dataset.

We performed experiments that fall into four broad categories: performance experiments, synthetic test cases, and real-world large scale experiments. All staging servers run on dedicated compute nodes with GPU accelerators.

Total number of cores	(16 to 128) + 16 + 16 = 48 to 160
No. of writer cores	$2 \times 4 \times (2\text{to}8) \times 4 \times 4 = 16$ to 128
No. of staging cores/nodes	16/1
No. of reader cores	16
Volume size	$64 \times 128 \times 64$ to $256 \times 128 \times 128$
In-staging data size (20ts)	80 to 640 MB
Workflow pattern	Write immediately followed by Read
Cycle for error detection	1 time step

Table 5.3: Experimental setup for performance tests.

5.3.1 Performance Experiments

In this subsection, we evaluate whether applying idle resources in data staging to perform silent error detection has any adverse affect upon the applications connected to the staging servers. In order to quantify the interference (if any) caused by the proposed technique, we measure the write performance of the staging servers with error detection enabled. We also record write-response time without error detection in staging as a baseline. The setup of these experiments is described in Table 5.3.

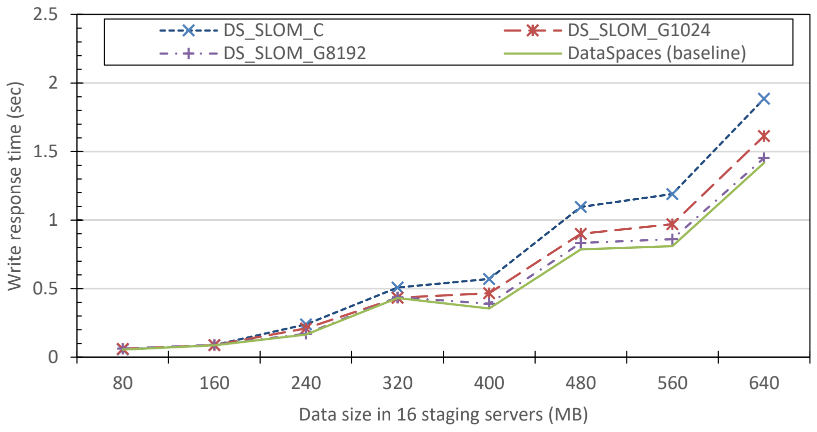


Figure 5.5: Comparison of the data write-response time for data staging with error detection on Titan. *DataSpaces:* Data staging write response time without error detection (baseline); *DS_SLOM_C:* Data staging write response time with error detection in CPU staging; *DS_SLOM_G1024/8192:* Data staging write response time with error detection in CPU-GPU hybrid staging under 1024/8192 GPU threads.

Figure 5.5 shows the cumulative write-response time for staging severs with error detection. We can see that error detection involve much lower performance penalties even at large dataset in data staging. For example, for CPU-based error detection in staging, the write response time overhead was no worse than 24.11% as compared to the same application without error detection. When we perform error detection in CPU-GPU hybrid staging, we see much lower overhead than the CPU-alone case. The write-response time increases by only 8.41% for the 1024 GPU thread case and 2.43% in the worst case for the 8192 GPU thread case as compared to the case of no error detection. Our approach shows good overall scalability and has minimal overhead as compared to the baseline.

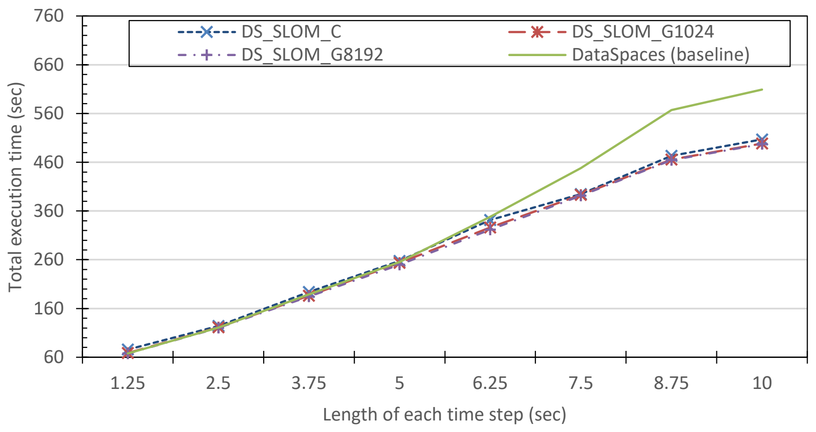


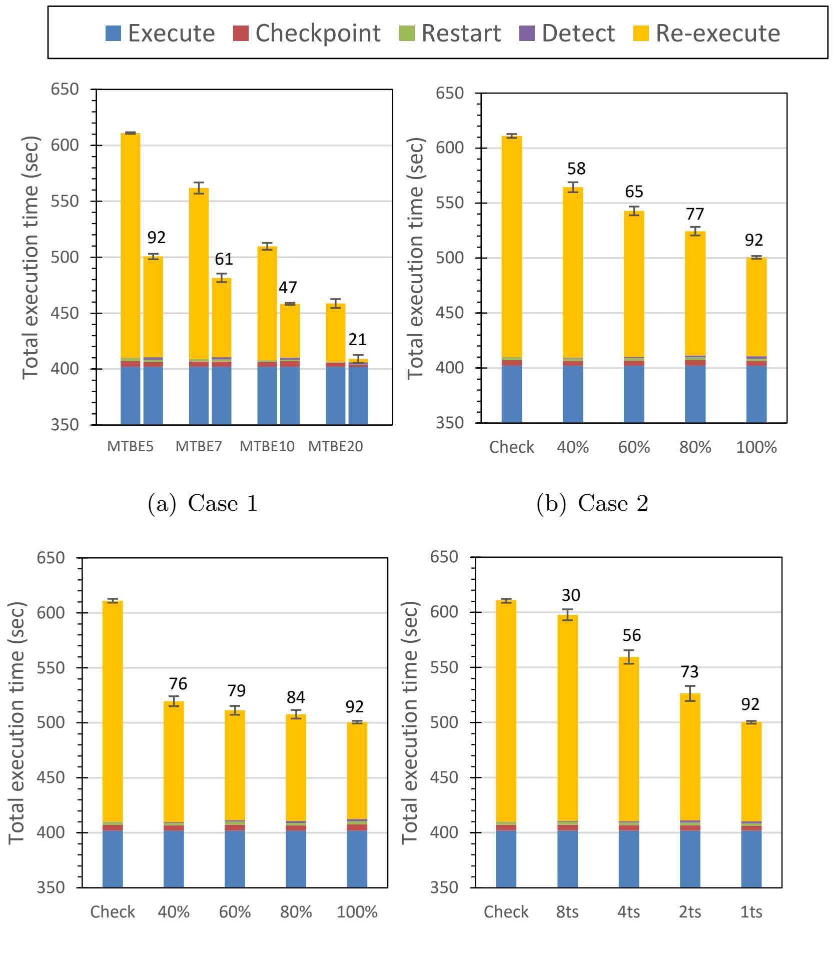
Figure 5.6: Comparison of the total execution time of the workflow under different error detection frequencies or length of each time step in data staging. Error detection is performed in each time step.

We also explore the overhead of error detection in staging over different workflow configurations. We plot the total execution time for a workflow with different error detection cycle in Figure 5.6. In this experiment, we perform error detection in staging in each time step and keep $MTBE = 5$ minutes and checkpoint cycle $T_c = 2.5$ minutes constant. We then vary the length of the time steps from 10 seconds down to 1.25 seconds. Figure 5.6 shows that, for larger than 5 second time steps, the benefit for error detection in staging becomes significant, with a decrease of up to 22.5% in the total execution time as compared to error detection in only the local component checkpoint. For the frequent error detection or smaller length of each time step(between 1.25 and 5 second), error detection in staging does not provide much benefit. At the extreme, for the 1.25 second time step case error detection can trigger an overhead of 11.7%.

5.3.2 Synthetic Test Cases

To better understand the performance of error detection in staging and its effectiveness, we also design synthetic cases and evaluate the total execution time of the workflow. Our synthetic experiments were performed on Titan. In these experiments, both synthetic simulation and analysis applications have their own local error detection and checkpoint component. For simplification we assume that the local error detection in checkpoint component has 100% *recall* and *precision*. We checkpoint every 10 iterations and before checkpointing dataset into the parallel file system, error detection is performed, guaranteeing that the checkpoint is error-free. Besides detecting silent errors in the local checkpoint component, the error detection is also performed through staging servers after the data is staged. We also assume that *precision* of error detection in data staging is 100%, and *recall* is linearly related with the proportion of staged data. Once a silent error is detected, the workflow is rolled back to the last checkpoint and re-executed. We ran the synthetic workflow under varying proportions of staged data to total data and different frequency of error detection in staging. Then, we measure the total execution time of the workflow. In these experiments, two significant soft errors were randomly introduced into the dataset within the 40 time steps, which corresponds to $MTBE = 5min$. We re-run our experiments for 50 times and report the total execution time as an average of these runs. The set-up of these experiments is described in Table 5.4. The experimental results are presented in Figure 5.7 followed by a detailed discussion and analysis of each.

1) **Case 1** - *Write the entire data domain in each time step under different* $MTBE$: Similar to simulation case 1 in Section 5.2.2, we vary $MTBE$ (occurrence of silent errors) between 5, 7, 10, and 20 minutes which corresponds to 100, 75, 50, and 25 silent errors respectively being introduced during a total of 50 runs. Since data staging can detect silent errors and roll back workflow early in each time, the method of performing error detection in staging can get a relatively better total execution time than the error detection in local checkpoint component under the high frequent $MTBE$. This improves the total execution



(a) Case 1

(b) Case 2

(c) Case 3

(d) Case 4

Figure 5.7: Breakdown of the total execution time (in seconds) for the workflows with checkpoint restart and error detection under 2 silent errors. *Execute:* Time spent in workflow execution; *Checkpoint:* Time spent to perform checkpointing; *Restart:* Time spent to restart workflow; *Detect:* Time spent performing error detection in staging; *Re-execute:* Time spent for re-executing the workflow from last checkpoint (correct error). The leftmost bar represents error detection in local checkpoint component only. Other bars represent error detection in both staging and local checkpoint components. Numbers on top of the bars indicate the total number of errors that was corrected by the staging component during 50 experiment runs.

time of the workflow by 18.1% for $MTBE = 5min$, 14.3% for $MTBE = 7min$, 11% for $MTBE = 10min$ and 10.1% for $MTBE = 20min$ respectively, as compared to error detection in local checkpoint component only under correspond $MTBE$ value.

2) **Case 2** - *Write a subset of the data domain and perform error detection in each time step*: This case is effectively a combination of simulation case 2 and 3 in Section 5.2.2. In this case and other subsequent cases, we inject a total of 100 errors during 50 runs of the experiment. Since different percentage of whole data-set is exchanged via staging framework, there is variance in the *recall* and overhead of error detection in data staging, which is based on proportion of subdomain size to the entire data domain. Thus, error detection in staging has an increase in detected errors when the amount of data in the staging area increases. Our approach reduced the total execution by around 7.6% for 40%, 11.3% for 60% and 14.2% for 80% subset data, as compared to the error detection in local checkpoint component only.

3) **Case 3** - *Write the entire data domain in multiple time steps and perform error detection in each time step*: This case differs from the previous case, where a certain percentage of entire data domain was written in a time step. In this case, although a subset in written in each step, data staging can stages entire data domain in between 2 to 3 time steps. This enables staging framework to have high cumulative *recall*. Therefore, error detection in staging reduces the total execution time (around 14.9% for 40%, 16.3% for 60% and 16.9% for 80% subset data) as compared to error detection in local checkpoint component only.

4) **Case 4** - *Write the entire data domain and perform error detection with different time step cycle*: Like simulation case 4 in Section 5.2.2, the entire data domain is written into the data staging periodically for every 1, 2, 4, 8 time steps separately. For error detection in staging, as the entire data is staged, a high *recall* of error detection is achieved. We can thus detect silent errors in the staging area and roll back the workflow immediately. Therefore, error detection in staging achieves a decrease of 2.2% for $8ts$, 8.4% for $4ts$, 13.8% for $2ts$ and 18.1% for $1ts$ in the total execution time relative to error detection in the local checkpoint component only.

Total number of cores	$96 + 16 + 16 = 128$
No. of writer cores	$6 \times 4 \times 4 = 96$
No. of staging cores/nodes	16/1
No. of reader cores	16
Volume size	$192 \times 128 \times 128$
In-staging data size (40 ts)	$960MB$
Workflow pattern	Write immediately followed by read
Local checkpoint, detection cycle	$10ts$

Table 5.4: Experimental setup for synthetic test cases.

5.3.3 Large Scale S3D Experiment

We perform large scale tests of our method using the combustion DNS-LES simulation/-analysis from the S3D combustion and analysis workflow [16] on Titan and compare it with error detection-only in the local checkpoint component. We also integrate error detection algorithm within S3D before checkpointing. Our staging-based error detection framework is tested using three different core counts (4416, 8832 and 17664) and corresponding grid domain sizes so that each core is assigned a spatial sub-domain of size $16 \times 16 \times 16$. In each time steps, 60%, 80%, 100% percentage of entire data passes through data staging separately. The total execution time of the workflow over 40 time steps is measured. Other core configurations, and data resilience setup are detailed in Table 5.5.

No. of cores	4416	8832	17664
No. of simulation cores	4096	8192	16384
No. of staging cores/nodes	256/16	512/32	1024/64
No. of analysis cores	64	128	256
Volume size	$256 \times 256 \times 256$	$512 \times 256 \times 256$	$512 \times 512 \times 256$
Data size (GB)	110	220	440
Checkpoint cycle	$10ts$	$10ts$	$10ts$
Detection cycle of staging	$1ts$	$1ts$	$1ts$
MTBE of silent error	$300sec$	$300sec$	$300sec$

Table 5.5: Configuration of core-allocations, data sizes, and data resilience for the three test scenarios on 4416, 8832 and 17664 cores.

Figure 5.8 and Figure 5.9 illustrate the experimental results for the total execution time of S3D coupled simulation and analysis application workflow for various scale in the presence of silent errors. We also measure the *precision* and *recall* of error detection in staging. For all tests, the *precision* always equals 100%, and the value of *recall* is shown on top of the bars in Figure 5.8 and Figure 5.9. For all error detection approaches, the workflow with error detection in CPU-GPU hybrid staging always shows lower execution time than the corresponding workflow running with CPU-based staging alone. Under error detection in CPU-only staging techniques with the presence of silent errors, error detection in staging reduces the total workflow execution time by up to 14.4%, 16.5%, and 18.9% as compared to error detection in only the local checkpoint component on 4416, 8832, and 17664 cores, respectively.

The same trend can also be seen in the CPU-GPU hybrid staging. The error detection in staging improves the total execution time by up to 14.1%, 22.6% and 15.4% as compared to a workflow with error detection in only the local checkpoint component. In addition, when error detection is performed in staging, it was observed that the greater the fraction of the dataset being put into staging and verified, the lower the total execution time in the presence of errors. These results demonstrate that error detection in staging achieves good overall scalability with small overhead for different processor counts and data sizes on extreme scale HPC systems.

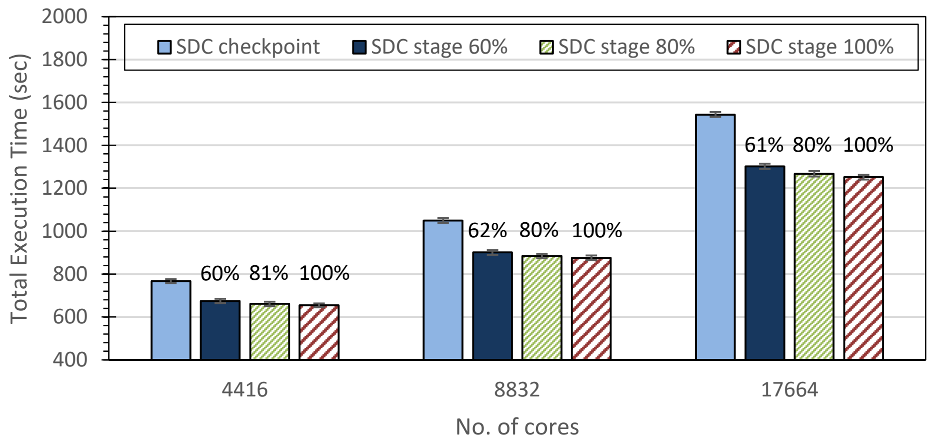


Figure 5.8: Comparison of the total execution time (in seconds) for the S3D simulation and coupled analysis workflow with error detection in CPU staging. Numbers on top of the bars indicate the *recall* of error detection in staging.

Another concern for error detection in data staging is about the extra energy consumption involved in utilizing a data staging area. For this paper, we did not explore energy optimization approaches for error detection in data staging, nor conduct experiments to evaluate energy consumption. Although evaluations of energy efficiency are beyond the scope of the paper, we can make some general statements about the energy consumption of our approach. Error detection in data staging introduces three main operations: data computation, data storage, and data movement. Transporting data across compute notes and storing it in DRAM devices increase the energy consumption [20]. Although it seems that adding error detection will increase the energy consumption for the workflow, if we consider that error detection in staging can decrease the total workflow execution time by eliminating substantial computation time in the presence of silent errors, we expect the energy consumption of the workflow to decrease.

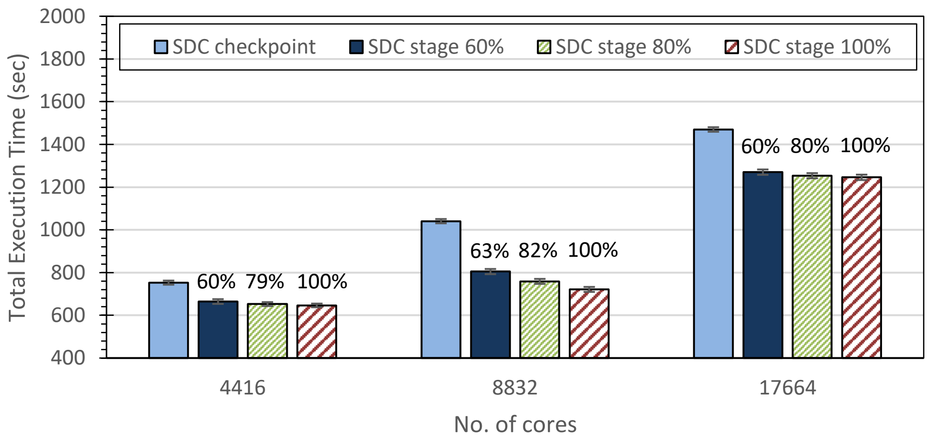


Figure 5.9: Comparison of the total execution time (in seconds) for the S3D simulation and coupled analysis workflow with error detection in CPU-GPU hybrid staging. Numbers on top of the bars indicate the *recall* of error detection in staging.

5.4 Related Work

Considerable efforts have been directed at developing techniques to detect and remediate silent errors. Process replication or redundancy techniques [29] can achieve a high *recall* and *precision* of detection with commensurately high compute and storage cost. Algorithm-based fault tolerance (ABFT) [10] can be very useful in decreasing the error detection cost and are specifically fit for detecting errors in linear algebra kernels using checksums. Detectors based on data analytics have been explored recently to serve as lightweight error detection methods[33, 7]. These approaches use interpolation techniques, such as time series prediction and spatial multivariate interpolation on the scientific dataset. In contrast to these efforts, we provide a framework for in-situ workflows for performing error detection. In addition, any error detection techniques can be easily integrated with our staging based error framework.

Since fail-stop failure is another significant risk for applications and checkpoint/restart has become a standard technique to address it, some research has combined checkpoint/restart with error detection approaches in order to deal with both fail-stop failure and silent errors. The study in [6] provides optimal resilience patterns to cope with fail-stop and silent errors. The research in [17] introduces an on-line ABFT based error detection approach and uses it in conjunction with checkpoint/restart to improve the time required to obtain correct results in an unreliable computing system. These approach are mainly designed for single applications and do not solve the problem of error propagation between coupled application. Our approach specifically targets the coupled scientific simulations and aims to eliminate error propagation via data verification in the shared abstraction, or a staging area.

5.5 Summary

This chapter presents a staging based error detection framework that uses idle computation resource to effectively detect silent errors for in-situ workflows. Then, this chapter introduces a design, implementation and deployment of this framework on top of the DataSpaces on Titan Cray XK7 at OLCF. Finally, This chapter have evaluated the effectiveness and performance of this framework through performance tests, synthetic tests and real world large scale S3D application runs. The experiments demonstrate that staging based error detection framework can effectively reduce the total execution time of scientific workflows when combined with checkpoint/restart.

Chapter 6

A Checkpoint/Restart with Data Logging Framework

6.1 Overview

In tightly coulped scientific workflows, simulation data is quickly shared and exchanged amongst different coupled applications for accelerating the overall scientific discovery. These simulation workflows running at extreme scales are providing new capabilities and opportunities in a wide range of application areas. However, due to the scales, coupling and coordination behaviors and overall data management complexities, they are also presenting new fault tolerance challenges that must be addressed before their potential can be fully realized. As demonstrated in Chapter 3, to efficiently maintain the crash consistency between coupled application components and enable diverse state-of-the-art fault tolerance approaches in the workflows have become significant and immediate challenges.

This chapter presents a workflow-level checkpoint/restart strategy for in-situ workflows. This framework employs a data/event logging mechanism to keep data consistency among application components during the failure recovery while decouple fault tolerance schemes between application components in workflows. Specifically, it performs data/event logging as soon as the data is written or read through the staging area. Also, it adapts global user interface to application components, which works with the queue based algorithm to record and replay data access events when preforming checkpointing and rollback recovery. In this way, the checkpoint/restart with data logging framework allows wide area fault tolerance schemes to be applied in workflows with flexibility and scalability, and minimize the interference between normal application components and the failed one when performing the recovery strategy.

The rest of the chapter is organized as follows. In Section 6.2, we introduce our checkpoint/restart with data logging framework. In Section 6.3, we implement and evaluate our approach using various synthetic scientific workflows. Section 6.4 provides details of various related work, and we conclude the chapter in Section 6.5.

6.2 Workflow-level Checkpointing Framework

In this section, we first propose our workflow-level uncoordinated checkpoint framework which is integrated with multiple checkpoint/restart strategies. We then extend the framework with other fault tolerance strategies such as process replication and ABFT, and construct a hybrid checkpoint approach. Finally, we design the global user interface for the framework which is implemented in open source data staging, DataSpaces.

6.2.1 Uncoordinated Checkpointing

To mitigate the data inconsistency issue discussed in Section 3.2.3, we introduce workflow-level uncoordinated checkpoint with data logging framework. In this framework, the workflow logs data transportation events and payloads between application components as it proceeds along the initial execution; without strong coordination, application components in workflows checkpoint their state independently. In case of application component failure, the workflow collects all its data/event log history, and enters the replay mode. Replay consists in following the log history, enforcing all data transportation events of the failed component to produce the same effect they had during the initial execution, and the corresponding data is re-provided to this process for this purpose. Therefore, the data dependency and consistency between coupled application components will keep during the replay mode. Once the history has been entirely replayed, the application component reaches a state that is compatible with the state of the other components in workflows, that can continue its progress from this point on.

Data Logging in Staging

One way to implement a data logging mechanism for workflows is to perform data logging in a data resilience staging area. Figure 6.1 shows a typical workflow with a uncoordinated checkpoint scheme combined with data staging. In this workflow, application components send the data communication requests to data staging. For data write requests, applications

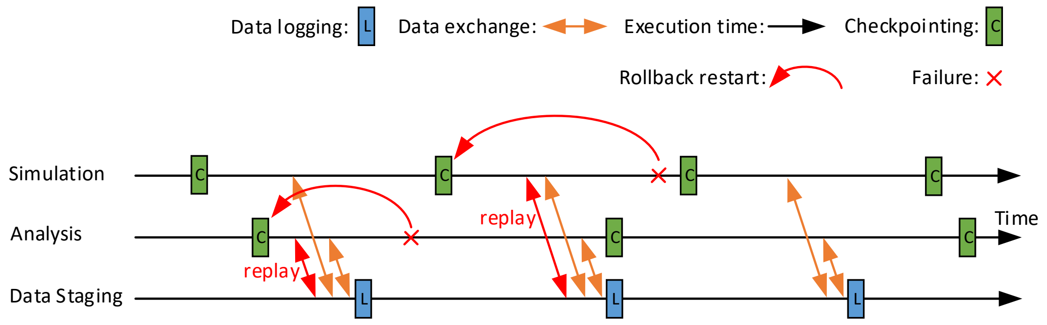


Figure 6.1: An illustration of uncoordinated checkpointing for a typical workflow with simulation, Analytic.

offload the data to the staging area for data transportation to coupled components later. For data read requests, applications receive the data from the staging area which are generated by coupled components in early time. Data staging logs the data communication requests and corresponding payloads, and records the fault tolerance events such as chechpointing and failure recovery during the initial execution. In case of failure, data staging switches to the recovery phrase. It cooperates with the application recovery scheme, and reproduces the data communication requests for the recovered component, which are determined by the log history in the initial execution. To guarantee the data availability in staging, the data staging can contain data resilience mechanisms such as data replication or erasure coding. It can also be integrated with the third part framework such as FTI [34] for data resilience.

Specifically, we employ a queue based data consistency algorithm in the staging area. Figure 6.2 illustrates a queue based data consistency algorithm for a coupled applications workflow. The workflow consists of two coupled applications, and in each coupling cycle (1 time step), coupled simulations a, b exchange the data through data staging. The data staging creates a event queue for each application, and pushes the data communication request events which related with the application into queue. In case of failure, the staging area will replay the events from the queue during the application recovery phase. In this example, simulation b failed and performs a rollback recovery at time step 7, and during time step 8 to 10, the staging area relays the events in the queue for the simulation b which are recorded from time step 5 to 7. At the end of checkpoint cycle($ts4$, $ts9$, $ts12$), data staging will clean the event queue and reload the following event from the front of queue. By maintaining the data request event queue, data staging can keep data consistency between coupled applications during failure recoveries.

As well as uncoordinated checkpoint with multiple checkpoint periods shown in Figure 6.2, this data logging mechanism can easily adapt to other checkpoint/restart strategies such as proactive checkpointing [11] and multi-level checkpointing [42] with only minor changes.

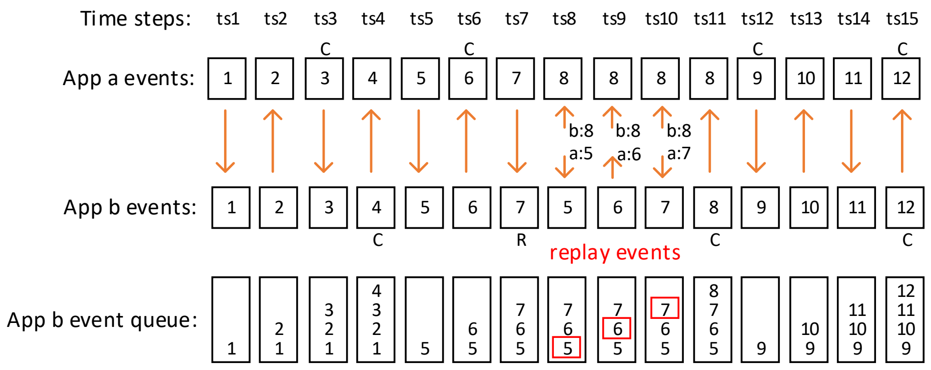


Figure 6.2: An illustration of queue based data consistency algorithm for a coupled applications workflow. Simulation *b* fails and performs rollback recovery at time step 7, then during time step 8 to 10, the staging area relays the events in the queue for the simulation *b* which are recorded from time step 5 to 7.

Storage Cost and Garbage Collection

To reducing the storage cost, a garbage collecting mechanism is provided in the staging area. Specifically, data staging servers periodically delete logged data which are related with previous checkpoint periods without data dependency to other application components, and only keep the latest version of data in the staging area.

6.2.2 Hybrid Checkpointing

Beside checkpoint/restart approaches, this workflow-level framework can also support the wide area fault tolerance mechanisms, and construct a hybrid checkpointing scheme for the workflow resiliency. Figure 6.3 illustrates a hybrid checkpoint framework which integrated with process replication and checkpoint/restart approaches. In this example, a simulation employs checkpoint/restart approach meanwhile the analytic uses process replication for resiliency. To make checkpoint/restart collaborate with a replication approach, during checkpointing periods, the data communication requests between these coupled applications will be logged in the staging area. If a failure happens in the application with checkpoint/restart, the application will rollback to last checkpoint place and re-execute through the latest checkpoint, and the data staging switches to recovery phase. The data logging in the data staging guarantees the restarted application can always get the correct version of data from the coupled application. Since the application with process replication can tolerate failures without rollback recovery, the failures will not trigger the data staging to switch to recovery phase and replay the events.

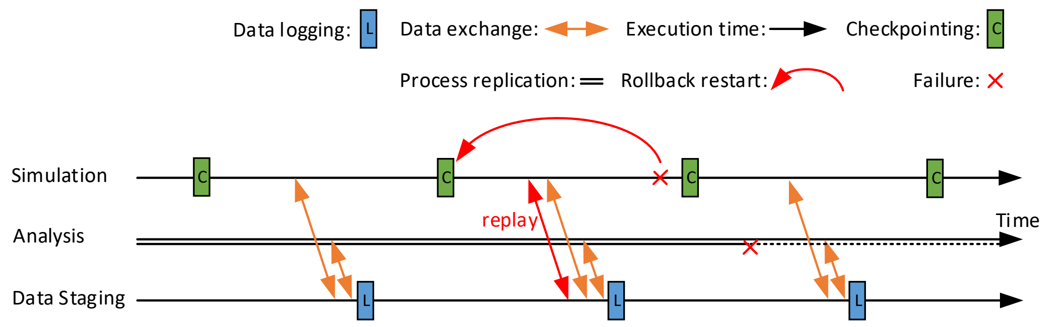


Figure 6.3: An illustration of hybrid checkpoint (integrated with a process replication) for a typical workflow with simulation, analytic.

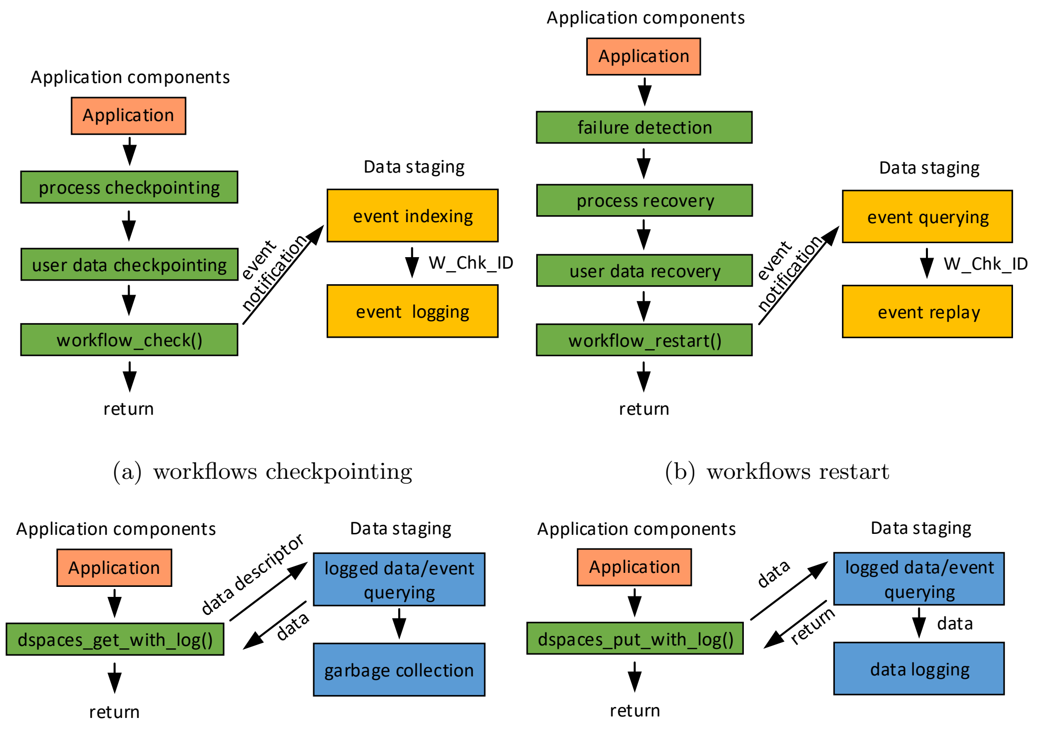
6.2.3 Global User Interface

In this section, we describe more details about global user interface, and present an example to illustrate how this interface can be used to integrate multiple application fault tolerance approaches to a workflow-level fault tolerance scheme.

As shown in Figure 6.4(a), When the application component performs checkpointing, it first saves process states and user-level data to the reliable storage devices. The checkpoints can be stored through a centralized parallel file system, assumed to be fault-free. Other options include storing the checkpoints in the node-local storage (such as NVRAM and SSD) or bust-buffer if the hardware architecture provides these devices. After that, the application component calls **workflow_check()** function, and notices a checkpoint event

workflow_check()	send a checkpoint event to data staging.
workflow_restart()	recover data staging client and notify the recovery event to data staging.
dspaces_put_with_log()	log data to data staging.
dspaces_get_with_log()	retrieve the logged data specified by geometric descriptor from data staging.

Table 6.1: User interface for checkpoint/restart in workflows.



(a) workflows checkpointing

(b) workflows restart

(c) data read with logging

(d) data write with logging

Figure 6.4: User interface for checkpoint/restart in workflows.

to data staging. When data staging receives the event notification, it creates a checkpointing ID: W_Chk_ID for this event, and then inserts it into the event queue. Since application components may have different checkpoint time spots, we assign an unique W_Chk_ID for each checkpoint event which refer to the same application component.

For the application recovery, it involves four key steps shown in Figure 6.4(b): failure detection, process recovery, data recovery, and data staging client recovery with event notification. To enable to recover applications from failures, application components will delete failed processes and recover the MPI communicator through ULFM [8] [9]; a proposed extension of the MPI standard which includes mechanisms for MPI applications to tolerate fail-stop failures. After that, the equal number of spare processes join the old communicator to construct the new one. An alternative approach is to spawn new processes instead of using processes from a previously prepared process pool, if this is supported by the job scheduler. After the failed application component recovered from the latest checkpoint spot, it calls **workflow_restart()** function. This function firstly initializes the data staging client, and tries to build an RDMA connection to data staging servers, and send the recovery event notification to the servers. After receiving the notification, the data staging servers will update the event queue, and generate a replay script for the recovered application.

During a workflow execution, when an application component tries to read the coupled data from data staging, it calls **dspaces_get_with_log()** function. This function sends data read request with the data descriptor to data staging. Based on the log history in the event queue, the data staging identifies whether the read request comes from a rollback execution or an initial execution of applications, and return the correct version of logged data. Finally, data staging performs garbage collection operation to erase those data which are no longer in use.

Similarly, when an application component tries to write the coupled data into data staging, it calls **dspaces_put_with_log()** function, and sends the data with the data descriptor to data staging. After receiving the request, data staging will query the events in the event queue. If the request is from initial execution, data staging will store data as the logged data otherwise omit the write request due to the redundant write request from the roll-back recovering application. Table 6.1 summarize the global user interface for application components to perform checkpoint/restart and data communication.

6.3 Experimental Evaluation

In this section, we describe the implementation details of workflow-level checkpoint framework and perform an experimental evaluation using synthetic benchmarks in the presence of failures.

Our workflow-level checkpoint framework is implemented on the top of CoREC [25], an open-source data staging with scalable data resilience. CoREC is a branch version of DataSpaces [21], and provides data resilience for a staging area in the case of both process and node failures while still maintaining low latency and sustaining high overall storage efficiency at large scales. The schematic overview of the runtime system is presented in Figure 6.5. In addition to modifying several existing components of CoREC for the integration, the system architecture introduces four key new components: Data Logging Component, Garbage Collection Component, Global User Interface, and Process/Data Resilience Component. The Data Logging Component stores, indexes and maintains the log data from coupled applications. Garbage Collection Component regularly cleans the unused historical log data. User Interface provides a set of checkpoint/restart and data logging interfaces for the applications in workflows. The Process/Data Resilience Component manages recovering applications from failures. This Component manages a spare process pool and implements the detection and handling of the process failures using ULFM, which offers a set of fault tolerance mechanisms for MPI applications.

6.3.1 Synthetic Experiments

Our synthetic experiments were performed on the NERSC Cori Cray XC40 system, and evaluated the write performance and memory usage of data staging with data/event logging. We also measured the total workflow execution time to evaluate the benefit from workflow-level checkpoint/restart framework in case of failures. To better understand its performance and effectiveness, we selected two test cases with common data access patterns and resilience

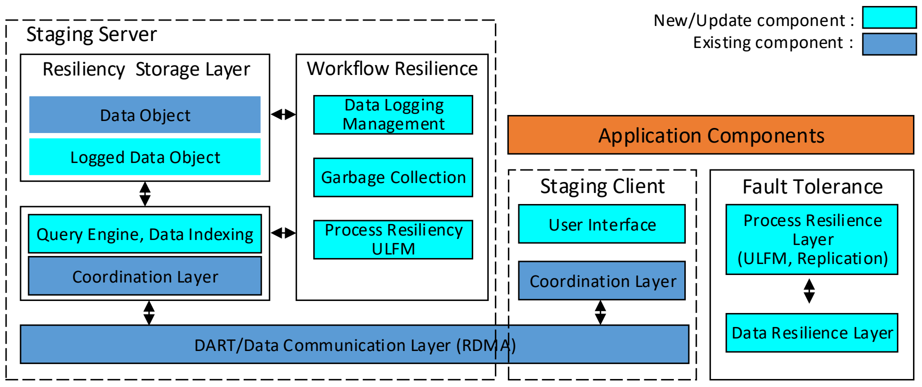


Figure 6.5: Implementation of workflow-level checkpoint framework

schemes used by real scientific workflows.

In each case, the simulation wrote the coupled data into the data staging, and the analytic read the data right after simulation write. Checkpoint/restart and/or process replication method were applied to the individual application to construct either uncoordinated checkpoint or hybrid checkpoint scheme for the entire synthetic workflows. In the synthetic workflows, simulation and analytic applications have different scales and resiliency requirements which correspond to checkpointing data to the parallel file system with different frequencies. For the process replication method, we use process duplication to tolerance one process failure. In case of failures, the application tolerated by checkpoint/restart will be rolled back to the last checkpoint place and re-executed from that point. For the application with replication scheme, it will be tolerated failures by switching the task from the failed process to the replicated process. We ran the synthetic workflow with different data access patterns and various frequencies of checkpointing. Then, we measure write response time and memory usage of data staging and the total execution time of workflows. In these experiments, a failure was randomly introduced into the application process within 40 time steps, which corresponds to $MTBF = 10min$. This simulates frequent failures on an extreme-scale supercomputer system. We compared our approach with two other fault tolerance mechanisms: global coordinated checkpoint (checkpoint application components in workflows coordinately, and restart them globally) as a baseline and individual checkpoint (individually checkpoint/restart application components without guarantee of correctness results) as the theoretical optimal lower bound. All experiments ran on the Cori, Cray XC40 system, a 12,076-node supercomputer located at the NERSC center at Lawrence Berkeley National Laboratory (LBNL). The set-up of these experiments is described in Table 6.2. The experimental results are presented in Figure 6.6 followed by a detailed discussion and analysis of each.

Total No. of cores	$256 + 64 + 32 = 352$
No. of simulation cores	$8 \times 8 \times 4 = 256$
No. of staging cores	32
No. of analytic cores	64
Volume size	$512 \times 512 \times 256$
Data size (40 ts)	$20GB$
Data access pattern	write immediately followed by read
Coordinated checkpoint period (ts)	4
Simulation checkpoint period (ts)	4
Analytic checkpoint period (ts)	5

Table 6.2: Experimental setup for synthetic test cases.

1) **Case 1** - *Write different subsets of the entire data domain in each time step*: In this case, 20%, 40%, 60%, 80%, 100% percentages of the entire data domain are exchanged between application components via data staging in each time step. Meanwhile, we perform checkpointing every 4 iterations for large simulation application, and 5 iterations for the small analytic application. With more data exchanging through data staging, both computation cost and storage cost for data/event logging increased. Data/event logging increased the write response time by 10%, 12%, 14%, 14%, and 15% as compared to the original data staging respectively. For the storage overhead of data/event logging in data staging, as shown in Figure 6.6(c), data/event logging increased the memory usage by 81% for 20%, 82% for 40%, 84% for 60%, 86% for 80%, and 86% for 100% subset, as compared to the original data staging's.

2) **Case 2** - *Write the entire data domain and perform checkpointing with different frequencies*: In this case, we write the entire data domain in data staging, and change the checkpoinnting periods from every 2 time steps to every 6 time steps. As seen in Figure 6.6(b), we got slight performance degradation when performing data/event logging in data staging. Data/event logging increased the write response time by maximum 14% as compared to original data staging under five different checkpoint frequencies. Since the less frequent checkpoint indicates the longer data/event queue size in the staging area, the

Total No. of cores	704	1408	2816	5632	11264
No. of simulation cores	512	1024	2048	4096	8192
No. of staging cores	64	128	256	512	1024
No. of analytic cores	128	256	512	1024	2048
Data size (40 ts)(GB)	40	80	160	320	640
Coordinated checkpoint period (ts)	8	8	8	8	8
Simulation checkpoint period (ts)	8	8	8	8	8
Analytic checkpoint period (ts)	10	10	10	10	10
$MTBF$(sec) / No. of failures	600 / 1, 300 / 2, 200 / 3				

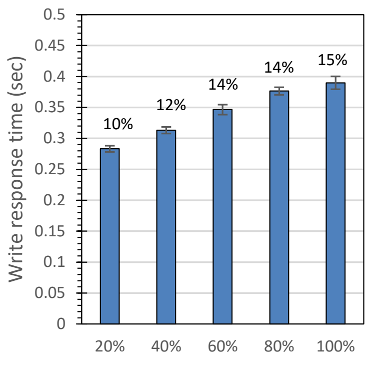
Table 6.3: Configuration of core-allocations, data sizes, and failure characteristics for the scalability test scenarios on 704, 1408, 2816, 5632, and 11264 cores.

higher storage cost can be expected. Therefore, as shown in Figure 6.6(d), the memory usage for data logging increases by 76% for $2ts$, 79% for $3ts$, 84% for $4ts$, 89% for $5ts$, and 97% for $6ts$ checkpoint period, as compared to the memory usage of original data staging.

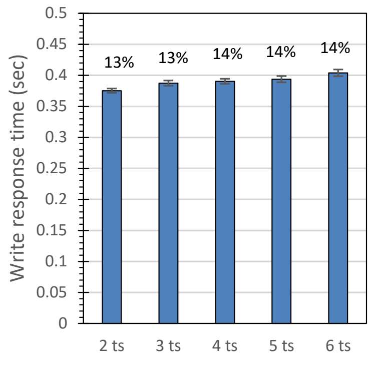
In these two cases, both uncoordinated checkpoint and hybrid checkpoint achieved nearly same execution time as individual checkpoint's which is theoretical optimal lower bound for the execution time of workflows with fault tolerance. Also, they achieve a decrease of 3.06% and 3.05% in the total execution time relative to global coordinated checkpoint in case 1. In case 2, as seen in Figure 6.6(e), we get similar performance improvement with case 1. The uncoordinated checkpoint and hybrid checkpoint reduce the total execution time around 3.15% for $2ts$, 3.28% for $3ts$, 3.26% for $4ts$, 3.05% for $5ts$ and 3.18% for $6ts$ relative to global coordinated checkpoint respectively.

In order to study the scalability of workflow-level uncoordinated/hybrid checkpoint, we also plot the total workflow execution time at different workflow scales and MTBF. The setup of these experiments is described in Table 6.3. Figure 6.7 summarizes the total workflow execution time in case of different numbers of failures (from 1 to 3) and scales (704, 1408, 2816, 5632, and 11264 cores). It can be seen that in the presence of multiple failures, workflow-level uncoordinated checkpoint reduced the total execution time by up to 7.89%, 10.48%, 11.5%, 12.03%, and 13.48% on 704, 1408, 2816, 5632, and 11264 cores scales in comparison to global coordinated checkpoint.

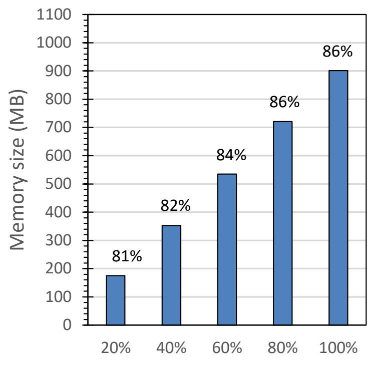
These results show that workflow-level checkpoint framework demonstrates good overall scalability and flexibility with small storage overheads for different data coupling patterns, fault tolerance schemes and processor counts.



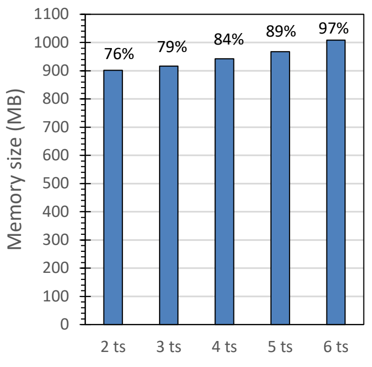
(a) Case 1 write latency



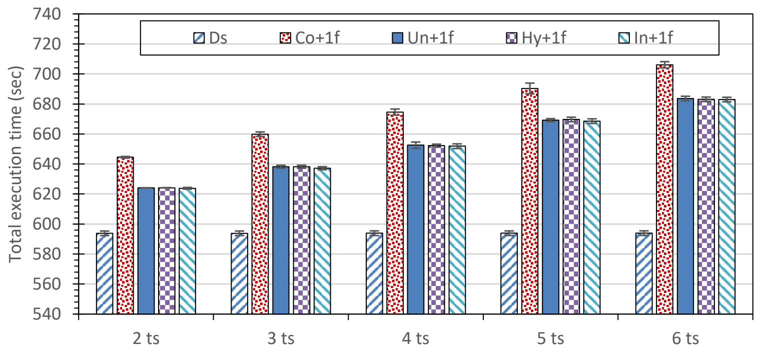
(b) Case 2 write latency



(c) Case 1 storage cost



(d) Case 2 storage cost



(e) Case 2 workflow execution time

Figure 6.6: Comparison of the cumulative data write response time, storage cost, and total workflow execution time using the synthetic workflow on Cori. *Ds:* The workflow with original data staging and failure free; *Co:* Global coordinated checkpoint/restart; *Un:* Uncoordinated checkpoint/restart; *Hy:* Hybrid checkpoint/restart with process replication; *In:* Individual checkpoint/restart; *+1f:* with one synthetic process failure. Percentages on top of the bars indicate the ratio of memory usage of data logging to the original data staging's, and the ratio of write response time delay of data staging with data logging to the original ones.

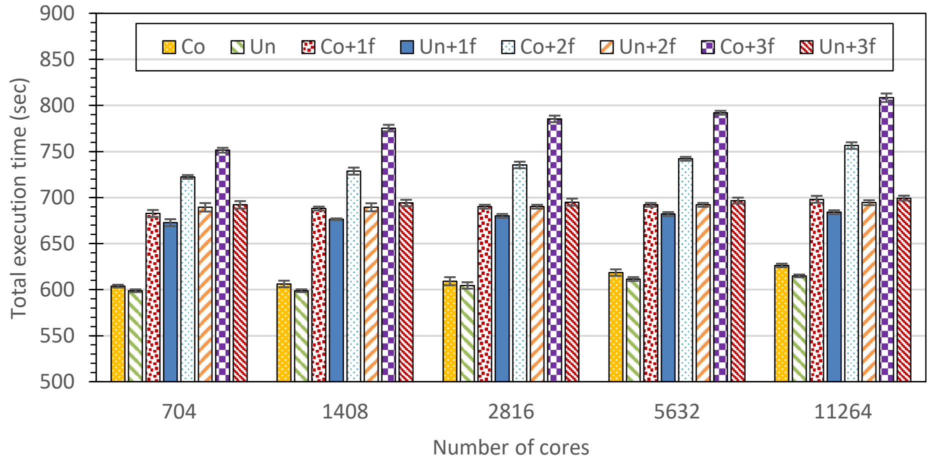


Figure 6.7: Summary of the total workflow execution time in case of failures (1, 2, and 3) and at different scales (704, 1408, 2816, 5632, and 11264 cores).

6.4 Related Work

Although various checkpoint/restart strategies [27] [12] [42] [31] [11] and process resilience techniques such as process replication or redundancy techniques [29] and algorithm-based fault tolerance (ABFT) [10] can effectually address the fail-stop failures in single application, there are limited research efforts on failure recovery for in-situ scientific workflows. The study in [40] exploits the reduction style processing pattern in analytic applications and reduces the complications of keeping checkpoints of the simulation and the analytic consistent. Research efforts in [41] use a synchronous two-phase commit transactions protocol to tolerate failures in high performance and distributed computing system. In comparison to these efforts, our checkpoint/restart with data logging approach specifically targets tight coupled in-situ workflows, and is more flexible, asynchronous and scalable.

6.5 Summary

This chapter presents a checkpoint/restart with data logging framework for tight coupled in-situ scientific workflows to enable diverse fault tolerance schemes to be used in workflows effectively and efficiently while maintaining data consistency. Then, this chapter introduces an implementation and deployment of uncoordinated and hybrid checkpoint framework on top of the DataSpaces on Cori, a Cray XC40 production system at Lawrence Berkeley

National Laboratory (LBNL). Finally, this chapter have evaluated the effectiveness and performance of our approach through synthetic tests. The experiments demonstrate that compared with global coordinated checkpoint, the uncoordinated checkpoint and hybrid checkpoint with data logging framework can effectively reduce the execution time of in-situ scientific workflows.

Chapter 7
Conclusions and future work

Cutting-edge in-situ workflows are generating and consuming data at an ever-growing scale. Data-staging frameworks and in-situ data process techniques have emerged as effective solutions for addressing data-related challenges at extreme scale and supporting in-situ workflows in high-performance computing (HPC) systems. However, reliability is an important issue that needs to be addressed in order to allow these workflows to continue scaling efficiently. The resilience challenge for extreme-scale in-situ workflows requires various hardware and software components in workflows are capable of handling a broad set of failures at accelerated fault rates. While the HPC community has developed various solutions, application-level as well as system-based solutions, the solution space of resilience techniques for in-situ workflows remains fragmented.

This thesis identifies and addresses key problems and requirements for in-situ scientific workflows. Specifically, this thesis presents CoREC and CoREC-multilevel, a scalable hybrid approach to data resilience for data staging frameworks that used online data access classification to effectively combines replication and erasure codes, and to balance computation and storage overheads. CoREC-multilevel can support different data resiliency techniques in order to satisfy the varying data resiliency requirements of multiple applications. Furthermore, utilizing lazy recovery and conflict-avoid encoding workflow optimizations, we reduced the interference due data-resiliency on the simulation/analysis components of the workflow. Secondly, this thesis presents a staging-based framework for detecting corruption that uses idle computation resource to effectively detect silent errors for in-situ workflows. As an illustrative example, we have demonstrated the use of an improved spatial outlier detection technique to achieve lightweight error detection with high accuracy. We have also provided a CPU-GPU hybrid staging architecture to minimize the impact of error detection on regular I/O operations on a data staging framework and interference with simulation/-analysis components of the workflow further when performing error detection. Finally, this thesis presents a checkpoint/restart with data logging framework for tight coupled in-situ scientific workflows to keep crash consistency and enable diverse fault tolerance schemes to be used in workflows effectively and efficiently. Specifically, we apply data logging in staging area to effectively decouple fault tolerance schemes between application components while maintaining data consistency among application components during the failure recovery. We have also provided a user interface for integrating this framework with application fault tolerance schemes. We evaluated the effectiveness, scalability and performance of the proposed programming interface and runtime mechanisms, through integration and experiments with synthetic and real-world in-situ scientific workflows.

In the future, this work can be extended in several directions.

- **Modeling and addressing failures for workflows in heterogeneous HPC system**: To sustain performance while facing always tighter power and energy envelopes, High Performance Computing (HPC) is increasingly leveraging heterogeneous architectures such as Burst buffer, General-purpose computing on graphics processing units (GPGPU), and Non-Uniform Memory Access(NUMA). However, this poses new challenges to efficiently model and address the heterogeneous architectures related failures in applications. In terms of in-situ scientific workflows, the required resilience management must support a wide range of different heterogeneous devices and programming models that target different application domains to provide end-to-end resilience solution for workflows.

- **Modeling and addressing silent errors in machine learning workflows**: This thesis focuses on in-situ scientific workflows which are composed by coupled scientific simulation and analytics. However, as machine learning becomes pervasive in high performance computing, it has found its way into scientific workflow domains (e.g.,knowledge discovery). Thus, the fault tolerance for scientific machine learning has grown in importance. Specifically, failures in machine learning workflows can have catastrophic consequences, and can occur due to silent errors, which are increasing in frequency due to system scaling. Meanwhile, Machine learning applications often possess uniquely inherent properties such as self-correcting behavior, due to their iterative convergent nature. Therefore, the new resilience frameworks for scientific machine learning workflows are needed to leverage these properties to achieve adaptability and efficiency by relaxing the consistency of execution and allowing silent errors to be self-corrected during workflow execution.

References

[1] L. Arturo, B. Gomez, N. Maruyama, and F. Cappello. Distributed diskless checkpoint for large scale systems. In *2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (CCGrid)*, pages 263–272, May 2010.

[2] G. Aupy, O. Beaumont, and L. Eyraud-Dubois. Sizing and partitioning strategies for burst-buffers to reduce io contention. In *Proceedings of the 33th IEEE International Parallel and Distributed Processing Symposium (IPDPS'19)*, pages 631–640, May 2019.

[3] L. Bautista-Gomez and F. Cappello. Detecting silent data corruption for extreme-scale mpi applications. In *Proceedings of the 22nd European MPI Users' Group Meeting (EuroMPI'15)*, September 2015.

[4] L. Bautista-Gomez, A. Gainaru, S. Perarnau, D. Tiwari, S. Gupta, C. Engelmann, F. Cappello, and M. Snir. Reducing waste in extreme scale systems through introspective analysis. In *Proceedings of the 30th IEEE International Parallel and Distributed Processing Symposium (IPDPS'16)*, pages 631–640, May 2016.

[5] J. Bennett, H. Abbasi, P.-T. Bremer, R. Grout, A. Gyulassy, T. Jin, S. Klasky, H. Kolla, M. Parashar, V. Pascucci, P. Pebay, D. Thompson, H. Yu, F. Zhang, and J. Chen. Combining in-situ and in-transit processing to enable extreme-scale scientific analysis. In *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, pages 1–9, Nov 2012.

[6] A. Benoit, A. Cavelan, Y. Robert, and H. Sun. Optimal resilience patterns to cope with fail-stop and silent errors. In *Proceedings of the 30th IEEE International Parallel and Distributed Processing Symposium (IPDPS'16)*, May 2016.

[7] E. Berrocal, L. Bautista-Gomez, S. Di, Z. Lan, and F. Cappello. Lightweight silent data corruption detection based on runtime data analysis for hpc applications. In *Proc. 24th International Symposium on High Performance Distributed Computing (HPDC'15)*, June 2015.

[8] W. Bland, A. Bouteiller, T. Herault, G. Bosilca, and J. Dongarra. Post-failure recovery of mpi communication capability: Design and rationale. In *International Journal of High Performance Computing Applications*, volume 27, pages 244–254, August 2013.

[9] W. Bland, A. Bouteiller, T. Herault, J. Hursey, G. Bosilca, and J. Dongarra. An evaluation of user-level failure mitigation support in mpi. In *Proceedings of the 19nd European MPI Users' Group Meeting (EuroMPI'12)*, September 2012.

[10] G. Bosilca, R. Delmas, J. Dongarra, and J. Langou. Algorithm-based fault tolerance applied to high performance computing. *Journal of Parallel and Distributed Computing*, 69(4):410–416, 2009.

[11] M. S. Bouguerra, A. Gainaru, L. B. Gomez, F. Cappello, S. Matsuoka, and N. Maruyam. Improving the computing efficiency of hpc systems using a combination of proactive and preventive checkpointing. In *Proceedings of the 27th IEEE International Parallel and Distributed Processing Symposium (IPDPS'13)*, pages 501–512, May 2013.

[12] A. Bouteiller, T. Ropars, G. Bosilca, C. Morin, and J. Dongarra. Reasons for a pessimistic or optimistic message logging protocol in mpi uncoordinated failure, recovery. In *2009 IEEE International Conference on Cluster Computing and Workshops*, pages 1–9, August 2009.

[13] P. Bremer, E. Bringa, M. Duchaineau, A. Gyulassy, D. Laney, A. Mascarenhas, and V. Pascucci. Topological feature extraction and tracking. In *Journal of Physics: Conference Series*, volume 78, page 012007. IOP Publishing, 2007.

[14] N. Budhiraja, K. Marzullo, and S. Toueg. The primary backup approach. In *Distributed systems*, page 2:199–216, 1993.

[15] F. Cappello, G. Al, W. Gropp, S. Kale, B. Kramer, and M. Snir. Toward exascale resilience: 2014 update. In *Supercomputing Frontiers and Innovations: an International Journal*, volume 1, pages 5–28, 2014.

[16] J. H. Chen, A. Choudhary, B. de Supinski, M. DeVries, E. R. Hawkes, S. Klasky, W. K. Liao, K. L. Ma, J. Mellor-Crummey, N. Podhorszki, R. Sankaran, S. Shende, and C. S. Yoo. Terascale direct numerical simulations of turbulent combustion using s3d. *Computational Science & Discovery*, 2(1), 2009.

[17] Z. Chen. Online-abft: an online algorithm based fault tolerance scheme for soft error detection in iterative methods. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP'13)*, August 2013.

[18] A. Cidon, R. Stutsman, S. Rumble, S. Katti, J. Ousterhout, and M. Rosenblum. Mincopysets: derandomizing replication in cloud storage. In *at the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2013.

[19] E. Deelman, T. Peterka, I. Altintas, C. D. Carothers, K. K. van Dam, K. Moreland, M. Parashar, L. Ramakrishnan, M. Taufer, and J. Vetter. The future of scientific workflows. *The International Journal of High Performance Computing Applications*, 32(1):159–175, 2018.

[20] M. e. M. Diouri, O. Gluck, L. Lefevre, and F. Cappello. Energy considerations in checkpointing and fault tolerance protocols. In *Proceedings of the Workshop on IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)*, June 2012.

[21] C. Docan, M. Parashar, and S. Klasky. Dataspaces: an interaction and coordination framework for coupled simulation workflows. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 25–36, 2010.

[22] C. Docan, M. Parashar, and S. Klasky. Enabling high-speed asynchronous data extraction and transfer using dart. *Concurrency and Computation: Practice and Experience*, 22:1181–1204, 2010.

[23] C. Docan, M. Parashar, and S. Klasky. Dataspaces: an interaction and coordination framework for coupled simulation workflows. *Cluster Computing*, 15(2):163–181, Jun 2012.

[24] C. Docan, F. Zhang, T. Jin, H. Bui, Q. Sun, J. Cummings, N. Podhorszki, S. Klasky, and M. Parashar. Activespaces: Exploring dynamic code deployment for extreme scale data processing. volume 27. Wiley Online Library, 2014.

[25] S. Duan, P. Subedi, K. Teranishi, P. Davis, H. Kolla, M. Gamell, and M. Parashar. Scalable data resilience for in-memory data staging. In *Proceedings of the 32th IEEE International Parallel and Distributed Processing Symposium (IPDPS'18)*, pages 105–115, May 2018.

[26] I. P. Egwutuoha, D. Levy, B. Selic, and S. Chen. A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems. In *The Journal of Supercomputing*, volume 65(3), pages 1302–1326, 2013.

[27] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys (CSUR)*, 34(3):375–408, September 2002.

[28] S. Feng, S. Gupta, A. Ansari, and S. Mahlke. Shoestring: probabilistic soft error reliability on the cheap. In *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems (ASPLOS'10)*, pages 385–396, March 2010.

[29] D. Fiala, F. Mueller, C. Engelmann, R. Riesen, K. Ferreira, and R. Brightwell. Detection and correction of silent data corruption for large-scale high-performance computing. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC'12)*, November 2012.

[30] B. Fitzpatrick. Distributed caching with memcached. In *Linux journal*, volume 124, page 5, August 2004.

[31] M. Gamell, K. Teranishi, M. A. Heroux, J. Mayo, H. Kolla, J. Chen, and M. Parashar. Local recovery and failure masking for stencil-based applications at extreme scales. In *High Performance Computing, Networking, Storage and Analysis (SC), 2015 International Conference for*, November 2015.

[32] S. Gao, B. He, and J. Xu. Real-time in-memory checkpointing for future hybrid memory systems. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, pages 263–272, Nov 2015.

[33] L. B. Gomez and F. Cappello. Detecting silent data corruption through data dynamic monitoring for scientific applications. In *the 19th ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP)*, pages 381–382, February 2014.

[34] L. B. Gomez, D. Komatitsch, and N. Maruyama. Fti: high performance fault tolerance interface for hybrid systems. In *2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 728–740, Nov 2012.

[35] S. Gupta, T. Patel, C. Engelmann, and D. Tiwari. Failures in large scale systems: long-term measurement, analysis, and implications. In *High Performance Computing, Networking, Storage and Analysis (SC), 2017 International Conference*, November 2017.

[36] S. Gupta, D. Tiwari, C. Jantzi, J. Rogers, and D. Maxwell. Understanding and exploiting spatial properties of system failures on extreme-scale hpc systems. In *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 37–44, June 2015.

[37] T. Herault and Y. Robert. Fault-tolerance techniques for high-performance computing. In *Fault-Tolerance Techniques for High-Performance Computing*, pages 12–24, 2015.

[38] A. Kougkas, H. Devarajan, X.-H. Sun, and J. Lofstead. Harmonia: An interference-aware dynamic i/o scheduler for shared non-volatile burst buffers. In *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, Sept 2018.

[39] A. Kougkas, M. Dorier, R. Latham, R. Ross, and X.-H. Sun. Leveraging burst buffer coordination to prevent i/o interference. In *2016 IEEE 12th International Conference on e-Science (e-Science)*, Oct 2016.

[40] J. Liu and G. Agrawal. Supporting fault-tolerance in presence of in-situ analytics. In *2017 17th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (CCGrid)*, pages 304–313, May 2017.

[41] J. Lofstead, J. Dayaly, I. Jimenezz, and C. Maltzahn. Efficient, failure resilient transactions for parallel and distributed computing. In *2014 International Workshop on Data Intensive Scalable Computing Systems*, pages 17–24, November 2014.

[42] A. Moody, G. Bronevetsky, K. Mohror, and B. R. d. Supinski. Design, modeling, and evaluation of a scalable multi-level checkpointing system. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC'10)*, pages 1–11, November 2010.

[43] B. Nie, D. Tiwari, S. Gupta, E. Smirni, and J. H. Rogers. A large-scale study of soft-errors on gpus in the field. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 519–530, March 2016.

[44] B. Nie, J. Xue, S. Gupta, T. Patel, C. Engelmann, E. Smirni, and D. Tiwari. Machine learning models for gpu error prediction in a large scale hpc system. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 95–106, June 2018.

[45] M. Parashar. Addressing the petascale data challenge using in-situ analytics. In *Proceedings of the 2Nd International Workshop on Petascal Data Analytics: Challenges and Opportunities*, PDAC '11, pages 35–36, New York, NY, USA, 2011. ACM.

[46] M. Parashar. Addressing the petascale data challenge using in-situ analytics. In *Proceedings of the 2Nd International Workshop on Petascal Data Analytics: Challenges and Opportunities*, PDAC '11, pages 35–36, New York, NY, USA, 2011. ACM.

[47] I. S. Reed and G. Solomon. Polynomial codes over certain finite fields. In *Journal of the Society for Industrial & Applied Mathematics*, volume 8(2), page 300, 1960.

[48] R. V. Renesse and F. B. Schneider. Chain replication for supporting high throughput and availability. In *in Proceedings of the 5th symposium on Operating Systems Design and Implementation OSDI'04*, pages 91–104, 2004.

[49] M. Snir, R. W. Wisniewski, J. A. Abraham, S. V. Adve, S. Bagchi, P. Balaji, J. Belak, P. Bose, F. Cappello, B. Carlson, A. A. Chien, P. Coteus, N. A. DeBardeleben, P. C. Diniz, C. Engelmann, M. Erez, S. Fazzari, A. Geist, R. Gupta, F. Johnson, S. Krishnamoorthy, S. Leyffer, D. Liberty, S. Mitra, T. Munson, R. Schreiber, J. Stearley, and E. V. Hensbergen. Addressing failures in exascale computing. *International Journal of High Performance Computing Applications*, 28(2):129–173, 2014.

[50] J. S.Plank, J.Luo, C. D.Schuman, L.Xu, and Z.-O. Hearn. A performance evaluation and examination of open-source erasure coding libraries for storage. In *the Seventh USENIX Conference on File and Storage Technologies (FAST)*, pages 263–272, Dec 2009.

[51] P. Subedi, P. Davis, S. Duan, S. Klasky, H. Kolla, and M. Parashar. Stacker: An autonomous data movement engine for extreme-scale data staging-based in-situ workflows. In *High Performance Computing, Networking, Storage and Analysis (SC), 2018 International Conference for*. ACM, 2018.

[52] P. Subedi and X. He. A comprehensive analysis of xor-based erasure codes tolerating 3 or more concurrent failures. In *Parallel and Distributed Processing Symposium Workshops and PhD Forum (IPDPSW), 2013 IEEE 27th International Symposium on*, April 2013.

[53] P. Sun and S. Chawla. On local spatial outliers. In *Fourth IEEE International Conference on Data Mining*, November 2004.

[54] J. T. Daly. A higher order estimate of the optimum checkpoint interval for restart dumps. *Future Generation Computer Systems*, 3(22):303–312, 2004.

[55] K. Tang, P. Huang, X. He, T. Lu, S. S. Vazhkudai, and D. Tiwari. Toward managing hpc burst buffers effectively: Draining strategy to regulate bursty i/o behavior. In *2017 IEEE 25th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, Sept 2017.

[56] U.S. Department of Energy, Office of Science. Exascale computing project. https://www.exascaleproject.org/exascale-computing-project/, 2018.

[57] L. Vervisch, E. Bidaux, K. N. C. Bray, and W. Kollmann. Surface density function in premixed turbulent combustion modeling, similarities between probability density function and flame surface approaches. *Physics of Fluids (1994-present)*, 7(10):2496–2503, 1995.

[58] D. Vogt, C. Giuffrida, H. Bos, and A. S. Tanenbaum. Techniques for efficient in-memory checkpointing. In *Proceedings of the 9th Workshop on Hot Topics in Dependable Systems*, pages 263–272, Nov 2013.

[59] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn. Ceph: a scalable, high-performance distributed file system. In *in Proceedings of the 7th symposium on Operating Systems Design and Implementation OSDI'06*, pages 307–320. Berkeley, CA, USA: USENIX Association, 2006.

[60] M. M. T. Yiu, H. H. W. Chan, and P. P. C. Lee. Erasure coding for small objects in in-memory kv storage. In *in Proceedings of the 10th ACM International Systems and Storage Conference (SYSTOR)*, May 2017.

[61] H. Yu, C. Wang, R. W. Grout, J. H. Chen, and K.-L. Ma. In situ visualization for large-scale combustion simulations. *IEEE Computer Graphics and Applications*, (3):45–57, 2010.

[62] J. Zawodny. Redis: Lightweight key/value store that goes the extra mile. *Linux Magazine*, 79, 2009.

[63] H. Zhang, M. Dong, and H. Chen. Efficient and available in-memory kv-store with hybrid erasure coding and replication. In *the Fourteenth USENIX Conference on File and Storage Technologies (FAST) for*, February 2016.