THE UNIVERSITY OF CHICAGO

APPLICATION-BASED FOCUSED RECOVERY (ABFR): CONVENIENT MANAGEMENT OF LATENT ERROR RESILIENCE USING APPLICATION KNOWLEDGE

A DISSERTATION SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCES
IN CANDIDACY FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

BY
AIMAN FANG

CHICAGO, ILLINOIS
AUGUST 2018

To my parents.

To See a World in a Grain of Sand

And a Heaven in a Wild Flower

– William Blake

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I would like to take this opportunity to express my gratitude to my advisor Prof. Andrew A. Chien. He always gives me patient guidance through my six-year Ph.D. journey. I have learned a lot from him, especially the scientific thinking and rigorous working. I will keep these in mind.

I would like to thank the other two committee members, Prof. Haryadi Gunawi and Prof. Shan Lu for their kindness to serve on my Master's and doctoral committee. They provided valuable suggestions on my work.

I sincerely thank many friends at UChicago for their support, especially Qinqing Zheng, Fan Yang, and Zhao Zhang. No matter where you are, I always wish you the best. And I want to thank all the members of LSSG team, especially Hajime Fujita, Nan Dun, Tung Hoang and Yuanwei (Kevin) Fang. They are not just my colleagues, but also mentors and friends. Many thanks to the department technical and administrative staff, especially Bob Bartlett, Sandy Quarles and Margaret Jaffey. I would not make through without their help and support.

At last, I want to thank my dear parents for their unconditional love, which supports me all my life.

ABSTRACT

Supercomputers continue to increase in scale and complexity to meet the demands of science and engineering. Exascale systems face high error rates due to increasing scale (10^9 cores), software complexity and rising memory error rates. Increasingly, errors escape immediate hardware-level detection, silently corrupting application states. Such latent errors can often be detected by application-level tests but typically at long latencies. Challenges for latent errors include determining when the error occurred, what data was corrupted, and how to recover efficiently. The predicted high error rates and latent errors are a critical problem that will increase the cost and may ultimately limit the scale of application science. However, existing fault tolerance approaches lack the support for latent errors. There is no general guidance to design latent error resilience.

This dissertation proposes a new approach called **A**pplication-**B**ased **F**ocused **R**ecovery (**ABFR**) for high-performance applications to execute efficiently in an environment with high error rates and latent errors. This approach exploits application knowledge to focus the recovery on only potentially corrupted data, achieving efficient and scalable latent error resilience. The two key ideas of ABFR are (1) clearly define the application knowledge needed for latent error recovery (as embodied in the four ABFR operators); (2) provide powerful runtime support to manage the complex recovery procedures, using the four application operators, without any other application programmer effort.

ABFR is a well-defined resilience framework that allows the application to pursue strategies exploiting a range of application semantics. Application designers can express their knowledge flexibly in four ABFR operators. ABFR is also an application-system partnership that provides a clear separation between application knowledge and the underlying system. Application designers implement four operators without concern for the underlying architecture and system details. The ABFR runtime implements the complex recovery procedure, including triggering and composing the operators, exploiting parallelism, and achieving load balance. Together, these ABFR properties support flexible application-based

resilience.

To demonstrate ABFR's generality, we apply it to three varied scientific computation archetypes (stencil, N-Body tree, and Monte Carlo particle transport). We design ABFR operators for each computation and evaluate the performance of ABFR. We measure latent error resilience performance for varied error rates. Results indicate ABFR significantly improves recovery performance. Specifically, ABFR reduces error recovery cost by 2.4x to 367x, recovery latency by 2.2x to 24x) and I/O cost up to 1000x. ABFR achieves efficient and scalable recovery at scale with high latent error rates for all three computation archetypes. Note that these results may be improved by more sophisticated application ABFR operators.

Overall, this dissertation demonstrates a new approach for efficient, scalable latent error recovery on large-scale systems. ABFR enables flexible application-based error resilience and provides sophisticated runtime support. As a result, applications are able to tolerate higher error rates and latent errors.

CHAPTER 1

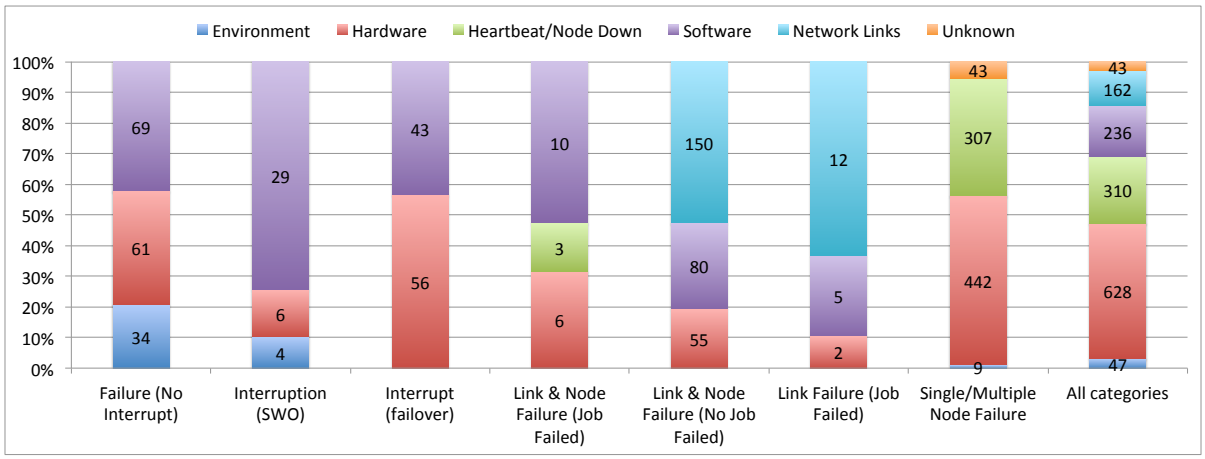
INTRODUCTION

1.1 Resilience Challenges at Extreme Scale Computing

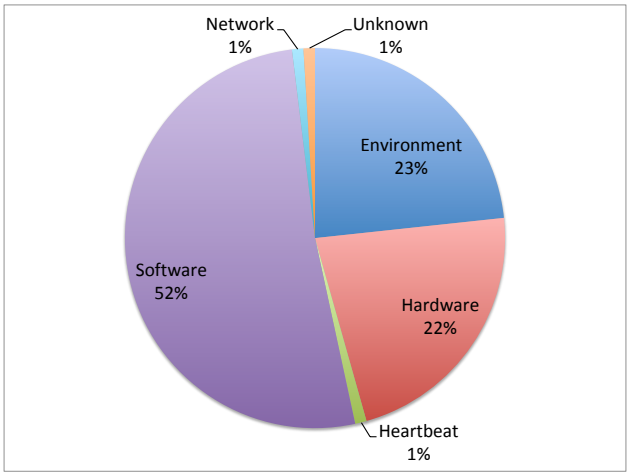
Large-scale computing is essential for addressing scientific and engineering challenges in many areas. Supercomputers continue to increase in scale and complexity to meet the demands of science and engineering. They typically consist of millions of components with growing complexity of software services (e.g. Summit [6], Sierra [11],TianHe-2 [7], Sunway TaihuLight [12]). However, progress from current top high-performance computing (HPC) systems (with tens of petaflops peak performance) to systems 1,000 times more powerful (e.g. exa-scale systems Summit, Aurora [2]) will encounter obstacles [89].

One of the main challenges to exascale is the likelihood of much higher error rates, resulting in frequent system failures or application producing incorrect results. System error rates in FITs (failures in 10^9 hours of operation) increase due to a confluence of many factors. First, the hardware failures are expected to be more frequent, due to radiation events, voltage drops, power leakage. Second, the software becomes more complex as with the hardware (heterogeneous cores, deep memory hierarchies, etc.). As a result, the software codes in larger scale (concurrency, network) are more error-prone.

Martino et al. [41] reported a study of failures on Blue Waters supercomputer [8], the Cray sustained petascale machine at Nation Center for Supercomputing Applications (NCSA), at the University of Illinois at Urbana-Champaign during a period of 261 days (March 1, 2013 to November 17, 2013). In this period, the system experienced 1,490 failures, namely a failure every 4.2 hours. The root causes of failures are classified into the following categories: hardware, software, missing heartbeat, network, environment, and unknown. Figure 1.1 shows how failure root causes distributed across the failure categories. Software and hardware are two dominant causes of failures. Software failures contributed to 53% of the node repair hours (hours required to repair failures), while hardware failures cause 23% of the total



(a)



(b)

Figure 1.1: Breakdown of the failure categories (a) distribution of the cumulative node repair hours, and (b) across hardware, software, network, unknown, heartbeat, and environment root causes. (Reproduced from [41].)

repair time. The top 3 software root causes are Lustre file systems, operating systems, and storage systems. These software systems are complex (more than 250k lines of code), and error prone in large-scale. Hardware failures come mostly from processors, memory DIMMs, and GPUs. Therefore, with the increasing scale and complexity, extreme-scale systems must perform correctly in face of errors both from hardware and software.

While already significant, errors are projected to increase significantly, producing mean time between failures (MTBF) as low as a few minutes [89, 27, 54]. Several studies projected

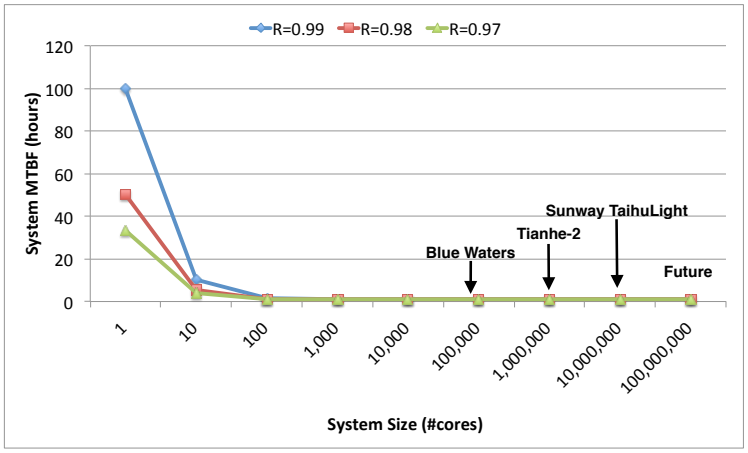


Figure 1.2: Projected system reliability. R: single core's one-hour reliability.

the evolution of the system MTTF (Mean Time to Failure) [65, 83]. To illustrate the critical consequence of such an increase, we depict the impact of growing failure rate and system size on MTBF in Figure 1.2. First, let R be a single core's one-hour reliability. An n-core system's MTTF is approximately $1/(1-R^n)$. Figure 1.2 shows the system MTTF for varied scale (system size) n and single core error rate R. With increasing system scale, the MTTF quickly drops to less than 1 hour. The future systems are expected to face severe failure condition. Applications without resilience support will not be able to finish the run or produce the correct results. Fault tolerance is strategic for high performance at increasing scale.

1.2 Latent Error Problem

Modern supercomputer hardware can detect and correct some errors [59, 77, 32], but some errors escape the hardware checks. These errors are **latent**, as they are only detected later when their corruption has spread. Latent errors are also known as **"silent data corruption" (SDC)** [55, 28, 53]. Such errors are a critical concern, since their data corruption threatens the correctness of computational results [86]. For example, the computational results of an application (climate, energy, national security, scientific discovery, etc.) can be corrupted due to hardware faults and returned to the users without any notification. Given that numerous

life and mission critical real-world decisions are made based on these corrupted results, it is easy to see the necessity of ensuring correctness. Some high-performance systems today already suffer from silent errors at a troublesome rate [88]. For example, the BlueGene/L system (106,496 nodes) experienced an L1 cache error (parity error) every $4-6$ hours [72]. The Cray XT5 at Oak Ridge National Laboratory experienced an uncorrectable double bit error on a daily basis [55]. Latent errors are more frequent than previously believed, and will become more so in larger scale systems.

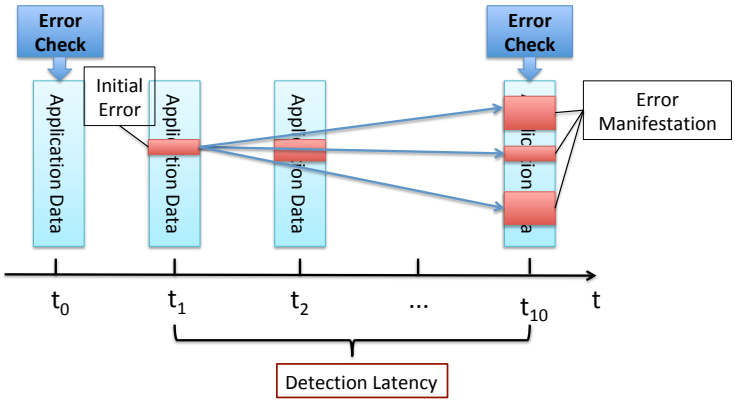
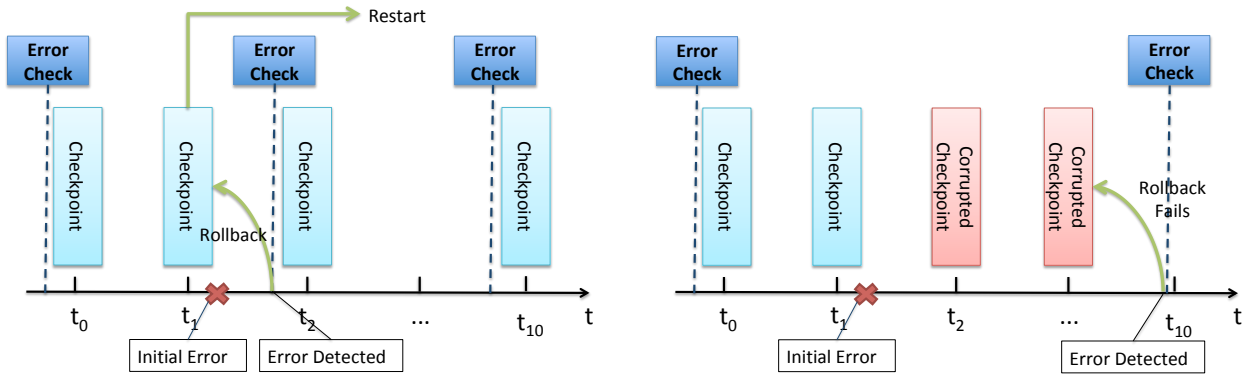


Figure 1.3: Latent errors: errors are detected some time after their occurrence and propagate to larger area of data space during this time period.

We focus on these **latent errors**, that escape system-level detection. These errors can only be exposed by sophisticated application, algorithm, and domain-semantics checks [63, 33]. Because these software checks are often computationally expensive, they are only run periodically to amortize their overhead. Figure 1.3 illustrates a latent error. As a computation progresses, the application data evolves from state to state. An error check is provided to examine the application data periodically (e.g. every 10 timesteps in Figure 1.3). An error occurs at time t_1 initially but the error check is not invoked at that moment. Instead, until time t_{10} the error check identifies one manifestation of the latent error. The time period $t_{10} - t_1$ is the error detection latency, during which time the error propagates through the application data space, corrupting more data.

Challenges for latent errors include determining:



(a) Immediate Error

(b) Latent Error

Figure 1.4: Checkpoint-Restart approach: (a) assumes immediate error detection, resilience achieved by writing periodical checkpoints and using rollback and restart; (b) does not work for latent errors as the checkpoints are potentially corrupted.

1. When the error occurred?

2. What data was corrupted?

3. and How to recover efficiently?

With predictions of higher error rates, these latent errors have the potential to limit the scale of application science. So good answers to these questions are essential for efficient latent error tolerance and recovery. There are two existing approaches to tackle the problem – system-level resilience and algorithm-level resilience.

System-level resilience. Checkpoint-Restart (CR) is a widely-used system-level fault tolerance technique, where resilience is achieved by writing periodic checkpoints, and using rollback and restart recovery. When an error is detected, applications rollback to the last saved checkpoint and restart computation. CR approaches rely on two critical assumptions: (1) the error is detected before creating the checkpoint; (2) the time to checkpoint and restart is << mean time between failure (MTBF).

CR style approaches do not work for latent errors, since they all assume immediate error detection. We explain two cases of error detection in Figure 1.4. CR performs an error check before writing the checkpoint in Figure 1.4a. The initial error occurred after timestep t_1 is

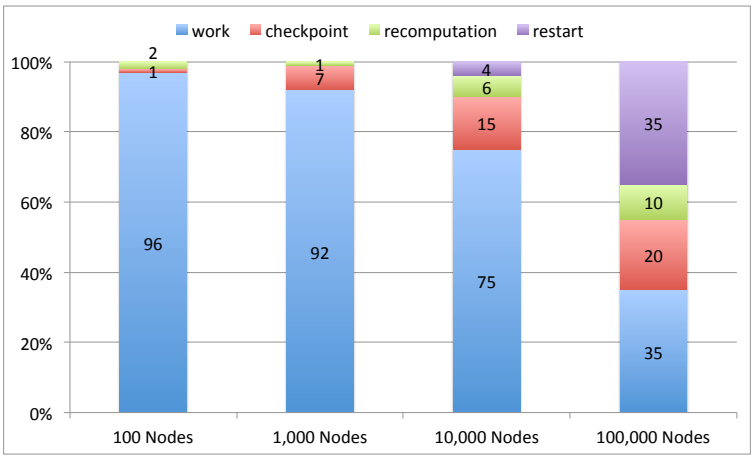


Figure 1.5: Efficiency of Checkpoint-Restart approach rapidly decays useful work for increasing node counts. (Reproduced from [55].)

captured before timestep t_2. CR rolls the application back to the checkpoint at timestep t_1 and restarts computation at that point. However, in case of complex latent error checks, their expense makes them invoked less frequently out of consideration for application runtime overhead. Even being applied at each step, the error check can still miss some errors due to the difficulty of error detection. In such a case (shown in Figure 1.4b), the error propagates, causing several checkpoints to be corrupted. Finally, at timestep t_{10}, the error manifestation is detected and the application uses the last checkpoint for recovery. But this checkpoint is corrupted! So the recovery will fail to correct the corrupted states. People may attempt to increase the checkpoint interval to eliminate the need of frequent error check and guarantee the correctness of last checkpoint. However, the longer checkpoint interval will dramatically increase the recovery cost since more work is lost. Another attempt is to create checkpoint more frequently. The issue of this solution is apparently the high cost. Previous work by Sandia shows that Checkpoint Restart approach rapidly decays the useful work of a 168-hour job for increasing node counts given MTBF of 5 years (see Figure 1.5). Only 35% of the work is consumed by computation while the remainder is spent on checkpointing, restarting and recomputation. CR approaches will become inefficient in extreme-scale systems with predicted higher error rates. There are several efforts on improving CR, including creating fast checkpoints and saving multiple checkpoints. We briefly introduce them and will discuss

their details in Chapter 2.

Fast CR: Rising error rates would require frequent checkpoints for efficient execution, and fortunately new, low-cost techniques have emerged [99, 29]. These fast CR approaches allow applications to create more frequent and cheaper checkpoints. Paradoxically, more frequent checkpoint increase the challenge with latent errors, as each checkpoint must be checked for errors as well. As a result, not all checkpoints can be verified, and latent errors escape into checkpoints. Thus, improved fast checkpointing does not help with latent errors.

Multi-level Checkpoints: It has been recognized that with latent errors, single checkpoints are insufficient, as such an error could corrupt the checkpoint. Researchers have proposed multi-level checkpointing systems and multiple checkpoint-restart approaches (MCR) [66, 17, 57, 20, 72]. These systems stores multiple checkpoints and exploits the storage hierarchy. They can be applied to latent errors. But the question which checkpoint is correct remains unclear. Therefore when an error check finds an error, these systems search back through the checkpoints, restarting, reexecuting, and retesting for error. This search is expensive, so new alternatives are desirable.

In summary, system-level approaches require least effort from application programmers but adds additional cost of runtime and resources. CR, fast CR and MCR do not work for latent errors and can not adjust to achieve efficiency.

Algorithm-level Resilience. Another direction to latent error problem is algorithm-level resilience. Algorithm-based fault tolerance (ABFT) is a long-standing technique that exploits algorithm and data structure properties to detect and correct errors. ABFT requires application developers to support resilience as part of their application codes, including detecting error and correcting error. ABFT has been extensively studied for widely used algorithms such as linear-algebra kernels [63, 85, 42, 33], including efficient schemes to correct single and double errors. ABFT can be used to address latent errors. ABFT shows the potential to tolerate latent errors more effectively by exploiting application knowledge. The primary lim-

itation of ABFT is its specificity to algorithm and data structures with each implementation done for a specific code base, execution environment, and system. ABFT has no guidance to design and reuse application resilience. No systematic approach has merged.

The existing attempts from system-level and algorithm-level did not solve the problem of latent errors. A general approach that provides systematic methodology and instructions for latent error resiliency is highly desired.

1.3 Thesis Statement

The Application-based Focused Recovery (**ABFR**) is an application-system partnership and enables application designers to flexibly express scalable, portable semantics-based application recovery. It is implemented by a sophisticated application-agnostic runtime that can achieve 100-fold reduction in error recovery cost, enabling scalable, high performance in exascale systems.

- **Semantics-based application recovery** This approach enables application designers to flexibly exploit application semantics such as algorithms and data structures to design customized fault tolerance. ABFR allows applications to easily experiment with different strategies, enabling portable, efficient forward, approximate, rollback, and more types of error recovery.

- **Application-agnostic runtime** The model defines a general framework for application recovery: four basic operators (check, inverse propagation, diagnosis, recovery) are implemented by application scientists without knowledge of the underlying architecture or runtime. The runtime triggers and composes these operators exploiting parallelism and conforming to data layout to recover efficiently.

- **Scalable, high performance** ABFR effectively focus recovery on where needed. The recovery cost is therefore independent of the number of processes and problem size. This approach achieves up to 24x reduction in recovery latency, 367x reduction in CPU

consumption, and 1000x less data read. It not only reduces execution time and I/O cost but also saves CPU throughput to support asynchronous execution that overlaps recovery and computation. As a result, applications are able to tolerate higher error rates in exascale systems.

1.4 Approach

The goal of this dissertation is to introduce and demonstrate a new approach for efficient and scalable latent error recovery. There are three key research questions taken into consideration for our approach.

1. **Challenge 1: How to achieve efficient and scalable latent error recovery?**

   As we discussed in the previous section, traditional CR approaches has two major issues that make them expensive: (1) the recomputation involves the full application and the recomputation length is the maximum potential error latency. However, in fact only a small subset of application data is potentially corrupted. It is possible to reduce the recovery effort if we can focus recovery on only corrupted data. Inspired by ABFTs, we can exploit application knowledge to identify potentially corrupted data and confine the recovery scope.

2. **Challenge 2: What application knowledge is needed?**

   Exploiting application knowledge for resilience is not a novel idea. The real question is how to provide general instructions for fault tolerance design. More specifically, what kind of application knowledge is required? Instead of customizing the requirements for each application, can we define the application knowledge needed generally across different types of computations?

3. **Challenge 3: How much effort is required from application designers?**

There are always debates on the question if we should ask extra programmer effort for resilience. If the sophisticated solution to high error rates and latent error problem desires application knowledge, can we minimize the effort? For instance, the solution should clearly define what application knowledge if needed, eliminating any ambiguities. Alternatively, the solution can provide additional support for complex recovery procedure to improve the performance.

We propose **A**pplication-**B**ased **F**ocused **R**ecovery (**ABFR**), an application-system partnership for efficient latent error tolerance. ABFR provides a systematic methodology for designing error resilience, exploiting application knowledge to focus recovery on corrupted data. It specifies the application knowledge needed for latent error resilience, capturing it in four operators from the application. This approach allows applications to easily experiment with different strategies. Application designers have the choice to implement operators exploiting a range of application knowledge. ABFR provides runtime support to organize operators and further improve their performance. It triggers and composes the operators exploiting parallelism and conforming to application data layout, etc. to recover efficiently. It effectively isolates the application from any need to understand runtime or fault-recovery mechanics. Therefore one implementation can be portable on different systems and even deployed across same classes of algorithms, alleviating much programmer effort.

To effectively focus recovery on potentially corrupted data, ABFR exploits Global View Resilience (GVR) [1] to create inexpensive versions of application states and use them for diagnosis and recovery. GVR demonstrated that versioning cost is as low as 1% of total cost for frequent versioning under high error rates [35]. A critical innovation of GVR is to create the ability to name application data across space (nodes, memory) and time (versions). This perspective is essential to conceiving and implementing the ABFR technique explored here. Specifically, GVR enables a range of flexible rollback and forward recovery, exploiting convenient access to versioned states. And, important for efficiency, with the global view of data and partial materialization, ABFR can flexibly access potential root causes across

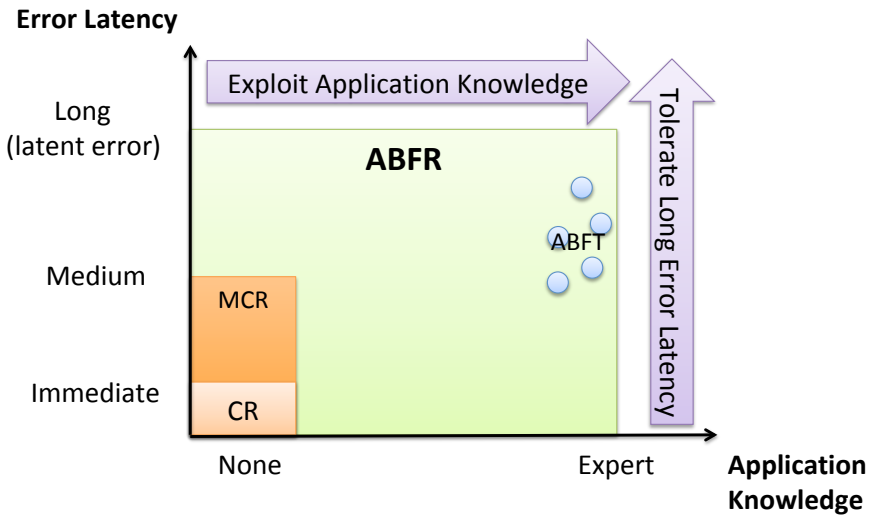


Figure 1.6: Applicability of resilience techniques across application knowledge and error latency space.

versions without reloading all the data, this allows focusing recovery to further reducing recovery cost.

Figure 1.6 illustrates the research space across application knowledge and error latency. ABFR expands the scope of feasible application-based resilience, enabling programmers to exploit application semantics flexibly and tolerate long error latencies. The traditional system-level approach Checkpoint-Restart is application-agnostic, and only tolerates immediate errors. Multi-level Checkpoint Restart approach extends the capability of CR by persisting more checkpoints. It is able to recover from medium-latency errors. However, in case of long-latency errors, the required iterative recovery incurs high overhead and risks encountering more errors during recovery. In Figure 1.6, we depict ABFT as a set of isolated solutions. That is because each ABFT algorithm and further each implementation of that algorithm is customized. Typical ABFT implementations rely on CR for storing checkpoints and recomputation, this cannot address errors with possible long latencies. The goal of ABFR is to provide the flexibility of exploiting application knowledge in designing resilience and equip them with the capability to tolerate long latent errors.

To demonstrate the generality of ABFR, we study three important application archetypes (stencil, N-Body tree, and Monte Carlo particle transport) that represent important algo-

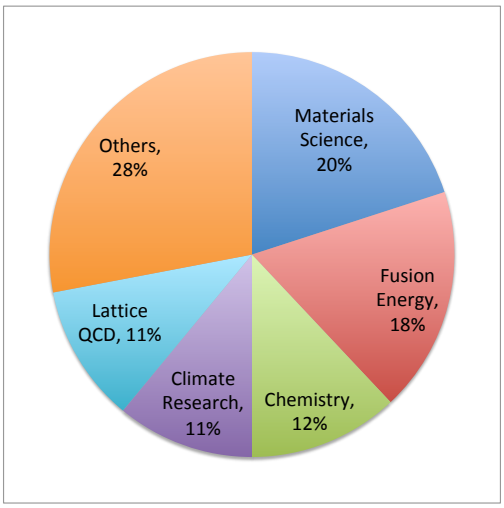


Figure 1.7: Top applications run on NERSC resources. (Reproduced from [10].)

rithmic structures and communication patterns widely-used in scientific computations. Figure 1.7 reports the workload distribution on National Energy Research Scientific Computing Center (NERSC) supercomputers. Stencil, N-Body tree, and Monte Carlo particle transport are widely used in the top three science categories. These archetypes are a challenging set, varying widely in communication pattern (fixed-regular to dynamic-irregular), data structure (dense array, tree, discrete), and as we will show, application recovery strategy. To illustrate the flexibility of ABFR provides, we discuss a few variants of ABFR operators for each archetype. We show that despite major differences in communication and parallelism structure, ABFR can be successfully applied. Some designs of ABFR operators are simple and straightforward, following the regular and predictable computation data flow and algorithm structures. Some knowledge exploited is intrinsically part of the application, already managed by developers, such as the error management. In some cases, when there is not enough information to follow, we can add a few data structures to enable ABFR operators. In all three archetypes, limited application knowledge is required.

To evaluate ABFR's performance, we use three large-scale production applications – Chombo (stencil), Gadget2 (N-Body tree), and OpenMC (MC particle transport). For each, we implement ABFR operators using the design described in Chapter 4. Our experience

show that less than 1.3% code changes are required for these production codes. We study ABFR's performance, varying error latency bounds that correspond to a range of error rates. We run experiments at substantial scale (up to 4,096 processes) on world famous supercomputers, projecting ABFR's performance and scalability. The results show that ABFR reduces recovery cost by 2.4x to 367x, recovery latency by 2.2x to 24x, I/O cost by over 1000x compared to CR approach, demonstrating ABFR achieves scalable recovery performance on large-scale systems. Especially, ABFR achieves the best performance at short error latency bounds. As higher error rates would shrink the optimal error checking interval (i.e. error latency bound), ABFR is even more valuable at high error rates. Note that these results may be improved by more sophisticated application ABFR operators. With such significant reduction in recovery cost, applications will be able to tolerate higher error rates in future systems.

1.5 Contributions and Outline

The specific contributions of this dissertation include:

- A new approach for latent error resilience – Application-based Focused Recovery (ABFR) – that defines the required application-semantics knowledge and enables application designers to conveniently express it, and achieves varied application-based resilience.

- An ABFR runtime that supports efficient latent-error tolerance for varied application types, providing runtime orchestration – automatically exploiting parallelism, data locality, and intelligent overlap to accelerate diagnosis and recovery, and improving scalability and performance.

- Insights into ABFR's generality through its application to three computation archetypes (stencil, N-Body tree, Monte Carlo particle transport), which are widely used in top 3 science categories in NERSC supercomputers. Four operator definitions for each

(and variations), reflect their parallel computation and communication structures, and studies demonstrate the flexibility and general applicability of ABFR.

- Parallel experiments at significant scale (up to 4,096 processes) show large ABFR performance benefits, reducing recovery cost by 2.4x to 367x, recovery latency by 2.2x to 24x, I/O cost by over 1000-fold. Enable efficient application execution at extreme-scale systems with high error rates.

The remainder of the dissertation is organized as follows. Chapter 2 introduces the background of system-level resilience and application-level resilience. We also provide an introduction on the Global View Resilience (**GVR**) library, the state-preservation foundation used in our ABFR experiments. In Chapter 3 we describe the ABFR approach which provides a general model for latent error recovery, four operator interfaces that define application knowledge a scientist must provide, and the ABFR runtime design that manages and accelerates recovery. Chapter 4 presents application of ABFR to three archetypes of large-scale scientific computations (stencil, N-Body tree, and Monte Carlo Neutron Transport), demonstrating ABFR's generality and applicability. In Chapter 5, we present measurements from three applications that evaluate ABFR performance for varied error latencies (error rates). We discuss how to automate ABFR in applications in Chapter 6. Finally, we summarize our work in Chapter 7, and suggest directions for future work.

CHAPTER 2

BACKGROUND AND RELATED WORK

High error rates predicted in exascale systems increase the application runtime cost and even jeopardize the correct execution of large high-performance computing applications, making fault tolerance a key feature to guarantee their completion and correctness. Much research has explored resilience in large-scale computations. A considerable number of researchers have studied error vulnerability: some of them focus on detection errors but rely on other methods to correct errors; others design error correction techniques. We classify these approaches into two categories: system-level resilience and algorithm-level resilience techniques. This Chapter first describes relevant prior work including Checkpoint Restart, Multi-level Checkpoint Restart and Algorithm-based Fault Tolerance and discusses limitations of them. We further review some existing attempts to solve latent errors. We show that no single solution has thoroughly addresses the issue of latent errors in large-scale. As last, we introduce the Global View Resilience (GVR) library, which we use to version application data in ABFR. These versioned intermediate application states are used to support ABFR's intelligent focused recovery. GVR's low-cost versioning enables flexible recovery for ABFR.

2.1 Checkpoint Restart

Applications without fault tolerance support stop working when a failure occurs. To tolerate fail-stop errors, a wide range of current production-quality applications use Checkpoint-Restart (CR) in application-level or system-level. To avoid having to restart the application from the beginning in case of random failures, the idea of CR is to save sufficient information periodically to stable storage such as parallel file systems, and restart the application to a previous point at which the information was saved. Such points are referred to as checkpoints, therefore the approach is called Checkpoint-Restart. System-level checkpoint technologies are broadly investigated [60, 75, 79]. System-level checkpoints are favorable as no changes

are needed in applications. Application-level checkpointing [34, 72, 20] requires programmers adding some functions in source codes to save application states. They are more flexible and lightweight compared to system-level technologies.

One of the challenges for checkpointing technologies is tuning the checkpointing frequency. The frequency of creating a checkpoint apparently affects the performance of applications. By saving checkpoints frequently, the application will lose less work in case of a failure and need less recovery effort, but the runtime is wasted writing frequent checkpoints. These checkpoints consume many resources (storage, network bandwidth), eventually wasting them in error-free runs. On the other hand, with less frequent checkpoints be taken, the application runs faster but will suffer greater work loss if a failure occurs

The problem of tuning the checkpoint frequency has been studied by many researchers. Young [98] and Daly [38] proposed a first order and a higher order approximation to the optimum checkpoint interval. The problem is formalized as an optimization problem of application runtime given conditions of estimated system failure rate, application error-free runtime, and checkpoint time. By the first order approximation, the optimal checkpoint interval τ_{opt} is given as:

$$\tau_{opt} = \sqrt{2\delta M}. \tag{2.1}$$

where δ is the time to write a checkpoint file and M is the mean time between failure. The approximation assumes the system exhibits Poisson single component failures. The resulted solution provides a simple formula to decide the checkpoint frequency for applications.

The success of CR approach crucially depends on two assumptions: (1) the time to checkpoint is much less than mean time between failure (MTBF); and (2) the error is detected before creating the checkpoint. It is reported that in 2009 parallel file systems are the dominant storage for storing application states [30]. Given the limited bandwidth of PFS, the checkpoint time was significant (often 15 to 30 minutes [72]). In-memory checkpointing was then proposed and demonstrated to be fast and scalable [99, 84]. We will introduce

some alternatives that further exploit non-volatile memories in next section.

We discussed in Chapter 1 that the CR approach generally assumes immediate error detection. That is, the last save checkpoint is assumed correct and will be used for recovery (see Figure 1.4a). However, when the error is not detected immediately, it will propagate to next checkpoints (see Figure 1.4b). Consequentially rolling back to the corrupted checkpoints will fail to recover. With rising error rates, it has been recognized that single checkpoint systems cannot handle latent errors because higher error frequency shrinks the optimal checkpoint interval [38], increasing the incidence of escaped error. Neither CR or fast-CR solves the problem of latent errors.

2.2 Multi-level Checkpoint Restart

At extreme scale, the cost of checkpointing to parallel file system (PFS) becomes prohibitive given that the bandwidth of PFS is limited. HPC researchers have proposed multi-level checkpointing systems and multiple checkpoint-restart (MCR) approaches [57, 20, 72].

Multi-level checkpointing creates multiple types of checkpoints that have different levels of resiliency and cost [92, 57]. Such systems exploit fast storage (DRAM, non-volatile RAM) and disks to maintain multiple checkpoints around. Inexpensive but less-resilient checkpoints are kept in fast, volatile storage, and expensive but most-resilient checkpoints in parallel file system. Multi-level checkpointing systems allow applications to make trade-offs between inexpensive checkpoints and expensive but resilient checkpoints, improving the efficiency and reducing I/O cost. Vaidya developed a Markov model for a two-level checkpointing system [92]. Gelenbe exploited the Markov model and derived formulas for system efficiency. Scalable multi-level Checkpoint System (SCR) [72] and Fault Tolerance Interface (FTI) [20] are two popular libraries that implement the multi-level checkpointing. Paradoxically, the frequent checkpoints that they enable do not help with latent errors.

With multi-level checkpointing systems, applications have multiple checkpoints available for recovery. The recovery scheme is so called Multi-Checkpoints Restart (MCR). MCR

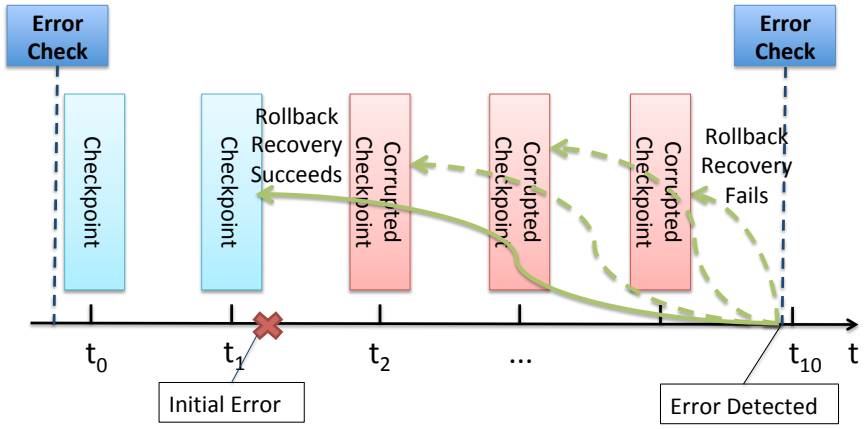


Figure 2.1: Multi-level Checkpoint Restart for latent errors: iteratively restarts from checkpoints, re-executes and tests.

still assumes applications recover from the most recent checkpoint when a failure occurs. Although there are a series of historical checkpoints, these multi-level checkpointing systems have no knowledge which checkpoint is correct. Therefore, when a latent error is detected, applications must search the checkpoints to find one that doesn't contain latent errors. The typical algorithm starts from the most recent checkpoint, reexecutes, and if the latent error recurs, repeats with the next older checkpoint. We present an example in Figure 2.1. The application uses multi-level checkpoint systems and an error check to verify the application state every 10 timesteps. The initial error occurs at time t_1. Since it is not captured immediately, it propagates to checkpoints created at t_2 to t_9. When the error manifestation is finally detected at t_{10}, the application first tries to rollback to the corrupted checkpoint at t_9 but the results of restart will fail again. It then recursively rolls back to checkpoints at t_8, t_7, and earlier states. Until restarting from checkpoint at t_1, the application can correct the erroneous states and continue execution. This blind search and global recovery incurs high overhead especially in case of errors with long latency. The rapid growth of recomputation costs make MCR unsuitable for latent errors. In short, the frequent checkpoints enabled by MCR and recent fast checkpointing technologies do not help with latent errors as they only increase the number of checkpoints within the error bound, that is, the number of potentially

corrupted checkpoints.

2.3 Algorithm-based Fault Tolerance

Algorithm-based fault tolerance (ABFT) arises as a promising alternative which can tolerate failures with low overheads. Huang and Abraham [63] proposed a checksum-based ABFT for linear algebra kernels to detect, locate and correct single error in matrix operations. The ABFT algorithm takes checksum vector or matrix as input operand and produces an encoded output matrix, which is then used to detect, locate and correct errors. Others extended Huang and Abraham's work for specialized linear system algorithms, such as PCG for sparse linear system [85], dense matrix factorization [42], Krylov subspace iterative methods [33]. In the works described above, ABFTs are mainly explored for matrix and vector based computations, and use techniques such as checksum and redundancy of information for fault tolerance.

Some studies explore focused resilience, but with immediate errors (not latent). Gamell et al. [56] studied local recovery for stencils. When a failure occurs, only the failed process is substituted with a spare one and rollbacks to the last saved state for the failed process and resumes computation. The rest of the domain continues communication. Their approach assumes errors do not spread across processes, limiting applicability to immediate errors. In contrast, ABFR can address long error latencies, including corrupted data across processes. Dubey et al. [43] explored local-recovery schemes for structured adaptive mesh refinement (AMR), exploiting the inherent structure within applications, recovery granularities can be controlled at cell, box, and level depending on failure modes. But this work also assumes immediate error detection. Dual-modular redundancy (DMR) detects errors by periodically comparing the computing results from two duplicated computing modules. DMR provides approximately 100% error coverage but incurs excessive overhead. Ren et al. [96] proposed Grid Sampling DMR (GS-DMR) to detect errors in stencil computations by comparing a subset of the results according to sampling on the grid data and exploiting error propagation

pattern on the grid. Widener et al. [95] explored machine learning techniques to detect memory failures. They attempted to apply supervised decision tree machine learning approach to evaluate the veracity of checkpoint and capture corruption in restart files.

ABFT can be more scalable [94] for some algorithms. It is potentially an appropriate fault tolerance technique with increasing failure rate [24]. However, ABFT has only be studied for limited set of algorithms, with each implemented for a specific code bases. There is no guidance on designing and reusing ABFT. It is not known if ABFT can be applied generally to all applications. Therefore, a more general approach is desired, with instructions on how to exploit application knowledge. ABFR addresses the specific issue that limit ABFT. ABFR is a general framework that provides clear instructions for designing application-based resilience. ABFR is an application-system approach, exploiting application knowledge for error detection, diagnosis, and recovery, and supplying sophisticated runtime support to ease the programming effort, and achieving efficient recovery from latent errors.

2.4 Latent Errors

Latent error problem is also known as **S**ilent **D**ata **C**orruption (SDC) in high performance computing community [22, 40]. SDCs are considered as one of the most serious challenges for HPC systems and applications. Silent errors are errors in application state that have escaped low-level error detection. At extreme scale, SDCs threaten the validity of computational results because there is no indication that there are errors during the execution.

Redundant computing has been proposed to detect and correct latent errors [67, 71, 54, 25]. The key idea is to replicate the computation at different levels (duplication, triplication or even more) and compare the results to detect errors. The different results indicate an error occurred. Triple Modular Redundancy or TMR [67], is the standard fault tolerance approach for critical systems, such as embedded or aeronautical devices [18]. However, triplication has a high cost, since two thirds of the computation resources are executing redundant work. To address the problem, many application-level error detection and correction techniques are

therefore proposed.

It has been recognized that exploiting application knowledge is a much cheaper method to identify errors. Many researchers proposed application-specific detectors. For example exploiting the smoothness of the evolution of a particular dataset in the in iterative methods (Berrocal et al. [23]) and predicted value change (Sharma et al. [87]). Their study showed that an interval of normal values for the evolution of the datasets can be predicted, therefore any errors that make the corrupted data point outside the predicted value change will be detected. Benson et al. [22] proposed an error check that uses a cheap auxiliary algorithm to repeat the computation at the same time with original algorithm, and compare the difference with the results produced by the original algorithm. Yim [97] explored the impact of transient faults in GPU devices on N-Body simulations and proposed utilizing physical properties to detect error.

While these efforts in error detection employed Checkpoint-Restart for recovery, ABFR can exploit them to far better effect.

2.5 Global View Resilience (GVR)

We use the GVR library [1] to preserve application data and enable flexible recovery. GVR provides a globally-visible, distributed array to applications, as in Global Arrays [74]. GVR uses versioned, distributed arrays to enable computational scientists to build portable, resilient applications. A critical innovation of GVR is to create the ability to name application data across space (nodes, memory) and time (versions) (see Figure 2.2). Applications create and name globally-visible arrays, and use them to save important data required for resilience. Applications control redundancy (per data structure), error checking, and recovery (exploit application semantics). With the space-time perspective of GVR, applications can easily manipulate data/versions for convenient use such as computation, analysis and recovery.

Key features of GVR include:

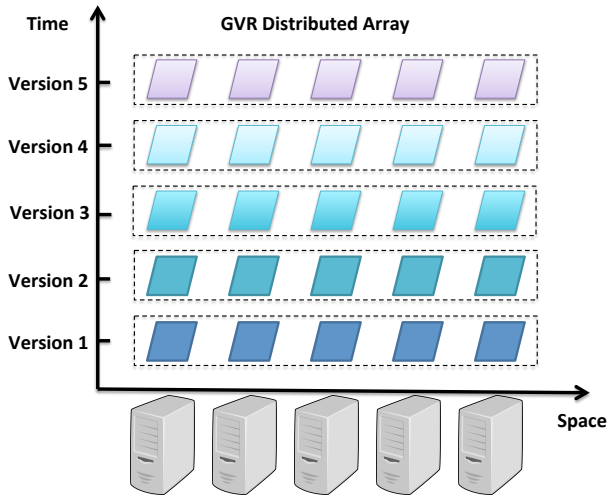


Figure 2.2: GVR Space Time Model: GVR enables naming and flexibly accessing globally-visible arrays across both space (nodes, memory) and time.

- multi-version distributed arrays that enable complex and latent error recovery,
- multi-stream versioning that gives the programmer control of when versions are created for an array, and
- unified error signaling and handling, customized per GVR distributed array, that enable algorithm-based fault-tolerance (ABFT) error-checking and recovery.

We introduce the basic GVR interfaces and key concepts such as global view, partial materialization and low-cost versioning that are essential to ABFR.

GVR Basic Interface. GVR's interfaces consist of two main parts: (1) basic data access, update, and version creation, and (2) error signaling and handling. GVR supports block-based access operations (`put`, `get`) on multi-dimensional arrays, synchronization operations (`wait`, `fence`), and accumulate operations (`acc`, `get_acc`). The basic GVR APIs are illustrated in Figure 2.3. These interfaces further introduce the concept of multi-stream, multi-version, and enable flexible recovery.

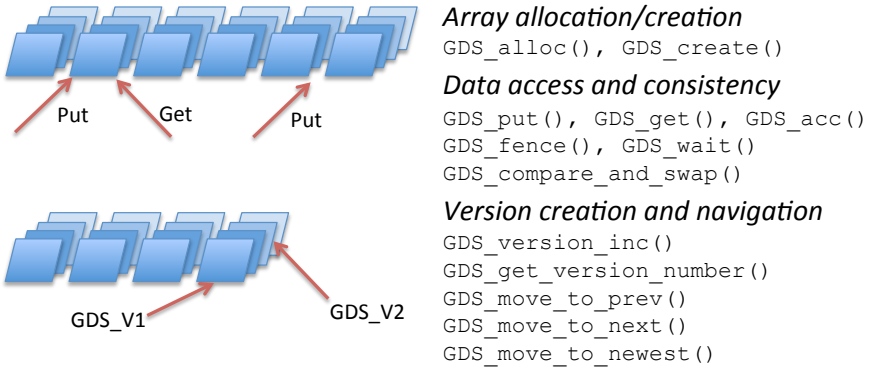


Figure 2.3: Distributed Arrays and Versioning on GVR

Global View Array and Partial Materialization. GVR distributed arrays each have a global name, but are distributed across multiple nodes [76, 74]. The global name supports flexible programming of irregular applications and, in the context of resilience, eases recovery programming when the number of physical resources have changed.

```
1  // Create a global array and name it gds_A
2  GDS_alloc(ndims,count,element_type,...,&gds_A);
3
4  // Put a range of value to global array gds_A
5  GDS_put(buffer, ..., lo_index, hi_index, gds_A);
6
7  // Get a range of value from global array gds_A
8  GDS_get(buffer, ..., lo_index, hi_index, gds_A);
```

Figure 2.4: Global View: naming and access

The name of GVR array is specified by user in `GDS_alloc` and `GDS_create` interfaces. An example is given in Figure 2.4: a large array is allocated and named *gds_A*. The size of this array can reach tera-bytes and is distributed across multiple nodes. With the global name, the parallel distributed applications can easily access any portion of the data by specifying the index or range of target data via `GDS_put` and `GDS_get` interfaces. In this way GVR enables flexible access to global arrays and partial materialization. This property distinguishes GVR from other CR style approaches, which always reload the whole checkpoints as they provide no knowledge or management of checkpoints. The capability of partial materialization is especially useful for ABFR's focused recovery. It is often the case

```
int main(int argc, char **argv) {
  int rank, size;
  GDS_init(&argc, &argv, ...);
  GDS_comm_rank(GDS_COMM_WORLD, &rank); // get the rank number
  GDS_comm_size(GDS_COMM_WORLD, &size); // get the size

  GDS_gds_t gds_A; // define a 2-dimension global array gds_A
  GS_size_t Asize[2] = {M, N}; // size of array gds_A
  GDS_alloc(2, Asize, ..., &gds_A); // allocate array

  double A[m*n]; // local data

  // computation loop
  for() {
    compute(A, ...); // do computation

    // put local value to global array
    GDS_put(A, offset, ..., gds_A);

    // create a new version
    GDS_version_inc(gds_A);

    // get value from several versions to local array
    for() {
      GDS_get(A, offset, ..., gds_A);
      GDS_move_to_prev(gds_A); // move to previous version
    }

    // move to the newest version
    GDS_move_to_newest(gds_A);
  }
}
```

Figure 2.5: GVR Code Example: create global arrays and access data across versions.

that only a subset of application data is corrupted by latent errors and needs correction. With GVR's partial materialization, ABFR can easily focus on potentially corrupted data and reduce the recovery scope. We give an example program in Figure 2.5, showing how to create global arrays and access a set of pieces of data across versions using GVR interfaces.

In summary, GVR's global view enables applications to easily create, version and restore (partial or entire) arrays. In addition, GVR's convenient naming allows applications to flexibly compute across versions of single or multiple arrays. GVR users can control where (data

structure) and when (timing and rate) array versioning is done, and tune the parameters according to the needs of the application.

Using GVR for Flexible Recovery. The ability to create multi-version array and partially materialize them, enables flexible recovery across versions. GVR has been used to demonstrate flexible multi-version rollback, forward error correction, and other creative recovery schemes on various computation archetypes (Monte Carlo Particle Transport, Preconditioned Conjugate Gradient solver, molecular dynamics, adaptive mesh refinement) [45, 51, 82, 100, 44]. Demonstrations include high-error rates, and results show modest runtime cost ($< 1\%$) and programming effort in full-scale molecular dynamics, Monte Carlo, adaptive mesh, and indirect linear solver applications [34, 35].

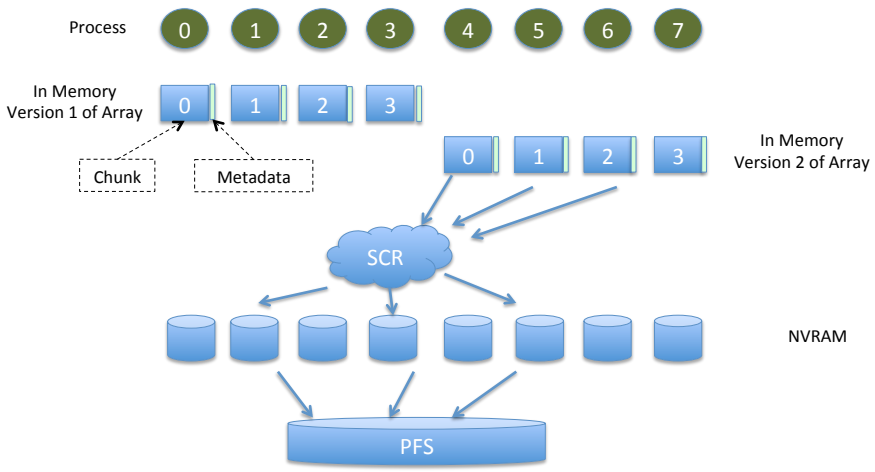
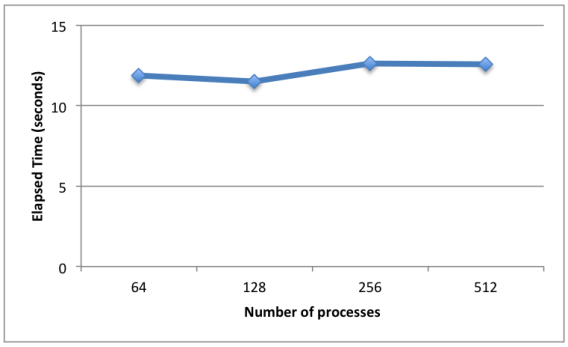


Figure 2.6: GVR leverages SCR to store versions across the storages.

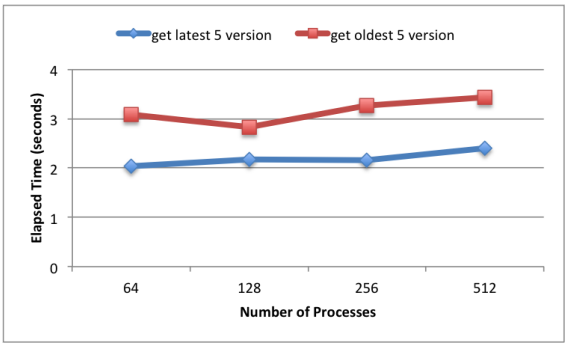
Feasible Low Cost Versioning. GVR exploits both DRAM and high bandwidth and capacity burst buffers or other forms of non-volatile memory to enable low-cost, frequent versioning and retention of large numbers of versions. As needed, local disks and parallel file system can also be exploited for additional capacity. GVR uses SCR [72] interfaces to store versions across the storage hierarchy. Essentially SCR serves as a file manager. The overview of the utilization of SCR in GVR is illustrated in Figure 2.6. In GVR's implementation, an

array is a set of chunks held by participating processes. In the versioning procedure, a version of the array is first created in memory by GVR. To flush a version to SCR, GVR uses SCR to create a file name for each chunk. GVR defines the suffix of the file name and records this metadata, so that GVR can map each chunk to the corresponding file. Given the file name, GVR writes chunks to files on local disks. SCR then manages these files and creates redundant copies exploiting various storages. The frequency at which SCR flushes files to the lower-level storage is determined by the runtime configuration. GVR manages metadata of versions and therefore can access them flexibly. For example, to read an element from an array, GVR first checks whether it is in memory. If it is in memory, GVR simply returns that element. Otherwise, GVR first calculates which chunk that element resides in. Parsing the metadata, GVR creates the suffix of the file name for the target chunk and calls SCR to obtain the full path of that file. This allows GVR to locate the actual file. Finally, GVR opens the file and reads the element using offset recorded in the metadata.

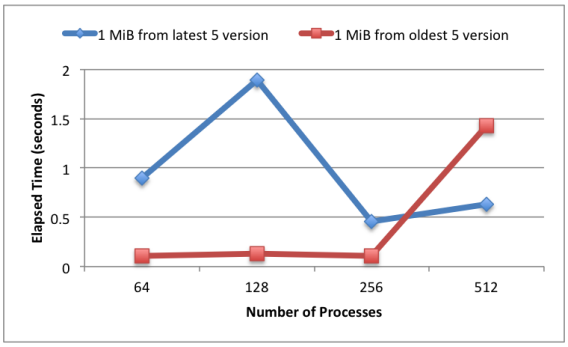
We measure the GVR versioning and read performance on two burst buffer systems. NERSC Cori [3] supercomputer provides 1.8 PB SSDs in the burst buffer, with 1.7 TB/s aggregate bandwidth (6 GB/s per node). The JUQUEEN supercomputer at Jülich Supercomputing Center [9] is equipped with 2 TB flash memory, providing 2 GB/s bandwidth per node. Figure 2.7 presents GVR's performance on CORI system. And multi-versioning performance studies on JUQUEEN [9] show that GVR is able to create versions at full bandwidth, demonstrating low cost versioning is a reality [46]. In this paper, GVR's global view and low-cost versioning enables flexible recovery for ABFR.



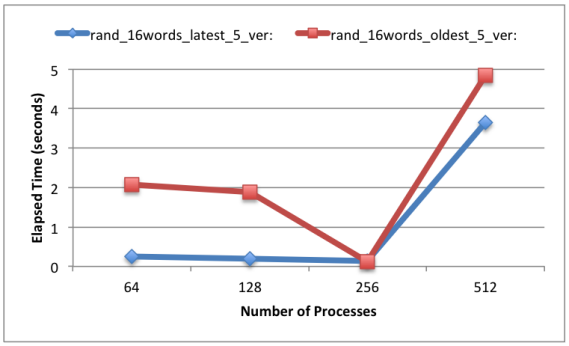
(a) GVR performance: create 10 versions on Burst Buffer.



(b) GVR performance: read 5 versions on Burst Buffer.



(c) GVR performance: read 1 Megabytes from 5 versions on Burst Buffer.



(d) GVR performance: read 16 words from 5 versions on Burst Buffer.

Figure 2.7: GVR performance on Cori burst buffer system.

CHAPTER 3

APPLICATION-BASED FOCUSED RECOVERY (ABFR)

In this chapter, we present our approach – **A**pplication-**B**ased **F**ocused **R**ecovery (ABFR) for achieving efficient and scalable latent error recovery. We begin with a motivating example, which reveals the opportunities for focused and intelligent recovery for applications in face of latent errors. With this inspiration, we present the overall design of ABFR and further introduce the two key components of our approach – ABFR operators and the ABFR runtime system.

3.1 Motivating Example

In Chapter 1, we define latent errors as errors that are detected with some latency after their occurrence. We look at a simple computation that encounters a latent error in Figure 3.1. This computation evolves on a 7x7 grid with an iterative manner. Each element on the grid is updated in every timestep. An error occurred at time t_1 and propagated to several other data elements due to algorithmic interaction between data elements. At time t_3, the error is detected by a verification scheme. We want to recover the correct state of this computation. If checkpoints are provided for each timestep, we can reload an earlier checkpoint before detection, for example, checkpoint at time t_2 and restart computation. However, this will produce incorrect results as the state at time t_2 is already corrupted by the error. This restart will produce a t_3 computation state that is incorrect, and possibly will fail the verification again. Failed at this checkpoint-restart attempt, the computation can iteratively retry from other earlier checkpoints, such at time t_1, producing the correct results if no error occurs during the retry process.

Such iterative rollback-recompute-verify approach is expensive but works for our simple example. However it will fail when the error rates increase or error latencies grow up. With higher error rates, the chances of encountering another error during recovery increases

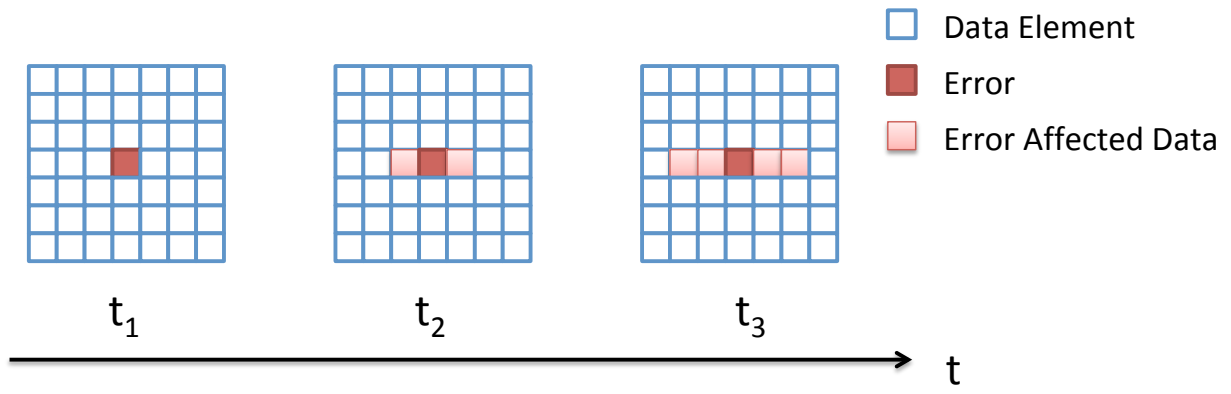


Figure 3.1: Motivating Example: Error propagated to some range of data. Recovery can be refined on error affected data instead of whole data space.

significantly and therefore the recovery will not succeed. In addition, Checkpoint-Restart style approach recomputes all the data, incurring tremendous computational cost.

One interesting observation from Figure 3.1 is that only a small range of data ($< 10\%$) is actually affected by the error. If we can accurately reduce the recovery scope to this data alone, the recovery cost can be reduced dramatically. The remaining question is how we can identify the corrupted data. If we treat the application as blackbox, we have no clue of data corruption. However, as the program designer, we master the algorithms and know how the application works. In this case, our example computation has following structure.

Algorithm 1 Example Computation Algorithm Structure

```
for K timesteps do
    for i: 1->m do
        for j: 1->n do
            grid[i][j] += grid[i][j-1] + grid[i][j+1]
        end for
    end for
end for
```

Each data element in the grid updates its value using its left and right neighbor value at each step. With this knowledge, we can deduce the error propagation path and identify data corruption area. To deduce error propagation, there is additional information required. First, we need the error location as the start point for deduction. In this case, the coor-

dinates of the detected error can be used and the set of corrupted data can be described using coordinates. Second, how far should we deduce the error propagation? One challenge with Checkpoint-Restart approach is that it can not decide which checkpoint is correct and therefore recursively try each checkpoint for recovery. Suppose we have an error check that can verify the state of the application, the state after verification is guaranteed to be correct. Such error checks are possible in application-level. Many scientific applications essentially need mechanisms to verify the results. These mechanisms can serve as error checks. Once equipped with such error checks, the error latency can be bounded by two consecutive error checks. As a result, we only need to deduce the error propagation for this latency bound and find out what data are corrupted and need to be recovered.

Walking through this simple example, we see that there are opportunities to efficiently recover from latent errors by refining recovery scope on only corrupted data. This strategy desires some application knowledge to help designers to figure out several key questions, including 1) where is the error, 2) how long (upper bound) does the error exist/propagate, and 3) what data is affected by the error. There is much application knowledge that can be exploited. It is important for application designers to determine which knowledge to use for efficient recovery. Our work proposes a framework that defines what application knowledge is needed from application designers and further provide runtime support, exploiting that knowledge to achieve efficient latent error recovery. In next section, we discuss the design of this framework.

3.2 Overview of ABFR Framework Design

Latent error resilience techniques depend on two elements – periodic checking for errors and persisting copies of application state. Error checking is done periodically to amortize the costly software checks required to detect latent errors. Figure 3.2 illustrates a typical execution model of application equipped with error checking and versioning.

When a latent error is detected, recovery can be done safely from a correct copy of

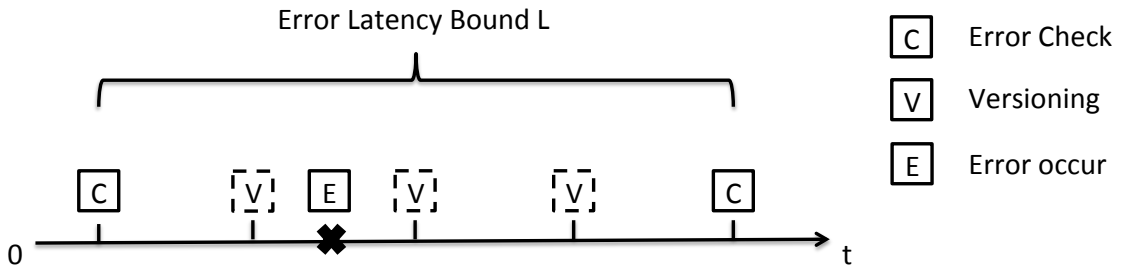
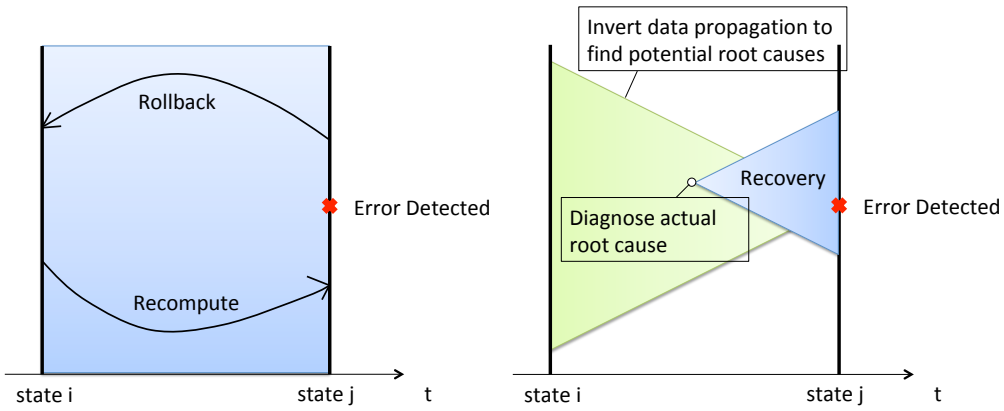


Figure 3.2: Applications with error checking and versioning. The interval between two consecutive error checks bounds the error latency.



(a) CR (b) ABFR

Figure 3.3: Checkpoint Restart (CR) vs. Application-based Focused Recovery (ABFR).

application state, captured at the last successful latent-error check. However, it is expensive, requiring extensive recomputation. Restarting the application and recomputing from that correct copy yields a checkpoint-restart (CR) style recovery for latent errors as shown in Figure 3.3a. Two problems with the CR-style approach make it expensive. Recomputation involves the full application and the recomputation length is the maximum potential error latency – the error checking period. In fact, the actual error latency may be shorter, and only a subset of the application data need to be recomputed.

Application-based focused recovery (ABFR) improves latent error resilience by exploiting application knowledge to reduce the recovery effort. As shown in Figure 3.3b, application knowledge can often enable reduction of the application scope (data breadth) as well as the execution time that must be recomputed or repaired. As a result, latent-error resilience

effort can sometimes be reduced by as much as two orders of magnitude. ABFR exploits application knowledge to reduce resilience cost , but the real contribution of ABFR is to enable applications to easily focus recovery effort. The two key ideas in ABFR are

1. **Clearly define the application knowledge needed for latent error recovery (as embodied in the four operators for ABFR).**

2. **Provide powerful runtime support to manage the complex recovery procedures, using the four application operators, without any other application programmer effort.**

With these two ideas, ABFR enables efficient recovery from latent errors. And, because the application programmer can change and improve their four operators (embedding more application knowledge), the application programmer remains in control with opportunity to refine and further improve latent error resilience.

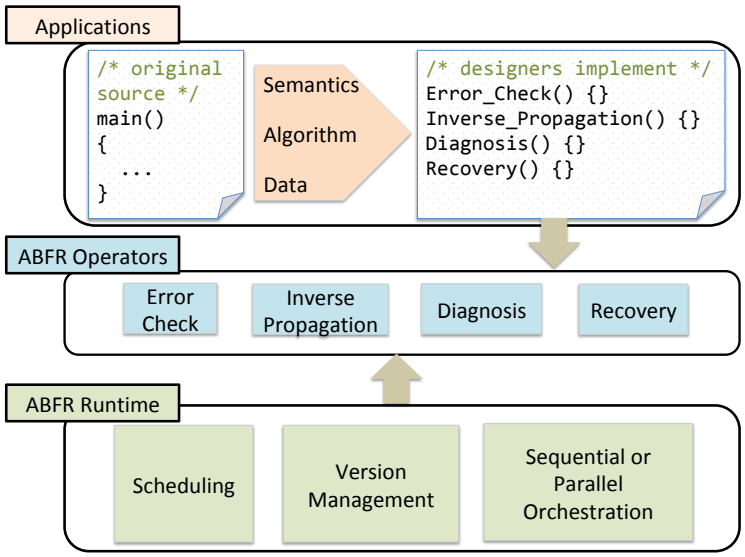


Figure 3.4: ABFR: Application, Operators, and Runtime

In Figure 3.4, we depict the relationship of the original application source, the application's four ABFR operators, and the ABFR runtime. The four operators extend the application for latent-error resilience, collectively encapsulating the needed application knowledge.

And the ABFR runtime employs sophisticated scheduling and orchestration for latent-error recovery using the four operators, achieving scalable, efficient execution. Thus, ABFR is an application-system partnership, and it defines the key knowledge required from applications – four operators: *error check*, *inverse propagation*, *diagnosis*, and *recovery*. The ABFR runtime then invokes the four operators, orchestrating them for efficient parallel execution to both focus recovery effort on the right application data and right execution period to efficiently recover from the detected error.

As shown in Chapter 4, the definition of each operator can exploit application-semantics to narrow potential error causes and minimize recovery effort – recomputation or repair (forward error-recovery). Because the recovery knowledge is localized and encapsulated in the operators, application programs can experiment with different recovery approaches easily. And further, they need not worry about underlying system details such as the versioning system and hardware architecture. We define the four basic operators then introduce the ABFR runtime design.

3.3 ABFR Operators: Encapsulate Application Recovery Knowledge

In any latent-error resilience system, the error check is invoked periodically; this period defines the "error latency bound" (see Figure 3.2). Error checks either succeed, or return the location and value of the error manifestation. ABFR versions application data between error checks to support efficient recovery. This versioning strategy is distinguished from that of traditional Checkpoint-Restart approach, which only saves a version of memory state after the error check. There are several reasons for our versioning choices. The error checks on application-level are usually expensive. First, frequently checking application states contradict the purpose of efficient and scalable application execution. Second, many applications naturally save intermediate results for future reference. At last, these intermediate states

can help refining recovery scope in case of an latent error. Of course versioning has some cost. The question remained to solve is how to determine the frequency of error checking and versioning. As with Young and Daly's [98, 38] effort to optimal checkpoint interval, we can also build analytical models with key factors taken into account and derive the optimal error checking and versioning intervals.

In this section, we define four ABFR operators and discuss the possible design choices by connecting to existing work. In Chapter 4, we will discuss detailed design for three exemplar scientific applications. At last, we present the ABFR operator interfaces.

3.3.1 Error Check Operator

Definition: If it detects an error manifestation, the error check returns the location and value of program data structure elements with corrupted values. Error checks are assumed to be expensive. [1] [2]

To address the problem of latent (or silent) errors, many application-specific error checks have been proposed. Much effort [63, 24, 85] has been expended on algorithm-based fault tolerance (ABFT), which exploits algorithms and application semantics to detect error and recover. As with these ABFT approaches, we utilize application knowledge, such as energy conservation, neighbor consensus, variation threshold etc., to design error checks.

3.3.2 Inverse Propagation Operator

Definition: given an error manifestation (value, location), return *all* data within the latency bound, that could have contributed to it. We call these potential root causes (PRC). Each PRC is a <data, time> pair. This set of PRCs must be complete, guaranteeing that recovery of this set of PRCs will correct any

1. Because latent ("silent") errors are complex to identify, this criteria is most inclusive, assuming expensive checks means that any improvements in checking can be used – cost is not a disqualifier.

2. Note that errors that cannot be detected are beyond the ability of any error recovery system to consider.

corrupted application data.

Given the error location and timing, application logic and dataflow – can be used to invert worst-case error propagation, identifying all PRCs in past that could have contributed to this error manifestation. Some applications have regular, local data dependencies or well known communication pattern, PRCs can be tracked step by step. While other applications have dynamic or complicate data flows, where using additional storage to record the dependencies can assist inverting error propagation. ABFR can be applied to a wide range of computations for which a conservative estimate is easily achieved. However, ABFR may be less effective for applications where errors quickly propagate and corrupt whole states. In such cases, a simplified version of ABFR can be deployed – detecting error and recovering all past states. In Chapter 4 we discuss the design of inverse propagation operator for three different communication patterns.

3.3.3 Diagnosis Operator

Definition: given a set of PRCs, do testing, version-comparison, or other computation to eliminate PRCs as candidates, reducing the number of PRCs. Reducing the number of PRCs reduces the recovery scope and cost.

The resulting PRCs of inverse propagation procedure can be a large set. To bound error impact more precisely, PRCs can be tested (diagnosis) exploiting deep application knowledge, eliminating many of the initial PRCs. For example, a simple diagnosis can be accomplished by recomputing intermediate states from versions and comparing to previously saved results. If the values match, the PRC can be pruned. With the support of GVR versions, such diagnosis can be parallelized and load balanced. PRCs across many different versions can be distributed to available tasks and tested concurrently, therefore further reducing recovery latency.

3.3.4 Recovery Operator

Definition: given a set of PRCs, use versions and recomputation or repair to recover the application state up to the current time. All corrupted states within the error latency bound are corrected.

Recovery is applied to the reduced set of PRCs and their downstream error propagation paths. A straightforward approach is recomputing and correcting the states of all PRCs. Other intelligent recovery techniques can also be deployed, such as forward error correction recovery [64, 34, 68] and approximate error correction recovery [36].

3.3.5 ABFR Operator Interfaces

```
1  struct abfr_error *abfr_check(void *data);
2  int **abfr_inverse_propagation(struct abfr_error *error, int
latency, int *nPRC);
3  int **abfr_diagnosis(void *data, GDS_gds_t version, int *nPRC,
int **PRCs, int step);
4  bool abfr_recovery(void *data, GDS_gds_t version, int nPRC, int
**PRCs, int step);
5  ABFR_status_t ABFR(void *data, GDS_gds_t version,
ABFR_diagnosis_mode_t dmode, ABFR_recovery_mode_t rmode);
```

Figure 3.5: Operator and other interfaces of the ABFR library

The operator interfaces are presented in Figure 3.5. The error check operator `abfr_check` takes the application state as input and returns one error (location and value) if error manifestations are captured. The `abfr_inverse_propagation` operator takes the error information and error latency as input. It sets the number of potential root causes and returns a set of PRCs. The diagnosis operator `abfr_diagnosis` takes the PRCs as input and returns the pruned set of PRCs. The recovery operator `abfr_recovery` takes the reduced set of PRCs as input and correct all the states within latency bound.

Combined with the semantic requirements described, these operators are all that need to be implemented by applications to achieve efficient application-based recovery. Programmers

Applications

```
main() {
   /* Create global array */
   GDS_alloc(dimension, size, type, priority, communicator, &gds);

   /* Computation Loop */
   main_loop() {

      /* Do computation */
      compute_stencils();

      /* Create Versions */
      if (Versioning_Point)
         GDS_version_inc(gds);

      /* Call ABFR to check error and trigger recovery */
      if (ErrorCheck_Point)
         ABFR(...);
   }
}
```

ABFR Operators

```
abfr_check(data) {
   /* verifies application data */
   data_check(data);

   /* return null or error */
   return abfr_error;
}

abfr_inverse_propagation(abfr_error, data, ...) {
   /* find and return PRCs */
   for (elem:data) {
      if(isPRC(elem) == true)
         abfr_PRCs.add(elem);
   }
   return abfr_PRCs;
}

abfr_diagnosis(abfr_PRCs, ...) {
   /* diagnose and return PRCs */
   for (PRC:abfr_PRCs) {
      if(diagnose(PRC) == false)
         abfr_PRCs.remove(PRC);
   }
   return abfr_PRCs;
}

abfr_recovery(abfr_PRCs, ...) {
   /* correct PRCs */
   correct(abfr_PRCs);
}
```

ABFR Runtime

```
ABFR(data, gds, diagnosis_mode, recovery_mode) {
   /* Call error check, if an error is detected, trigger other operators */
   abfr_error = abfr_check();
   if (abfr_error == NULL) return;

   /* Find potential root causes */
   abfr_PRCs = abfr_inverse_propagation(...);

   /* Manage versions and idle resources for parallel diagnosis */
   if(diagnosis_mode = ABFR_DIAGNOSIS_PARALLEL)
   {
      collect_resources();
      distribute_workload();
      reduced_PRCs = abfr_diagnosis();
   }
   /* Correct corrupted states */
   if(recovery_mode = ABFR_RECOVERY_PARALLEL)
   {
      distribute_workload();
      abfr_recovery();
   }
   /* Manage versions */
   GDS_move_to_newest(gds);
}
```

Figure 3.6: Example application codes using ABFR

call the ABFR() function to implement periodic error checking. When an error is detected, the ABFR runtime triggers ABFR recovery by calling the application-defined ABFR operators.

Figure 3.6 illustrates a snippet of application codes using ABFR interfaces. The application calls GVR function to allocate and name global arrays to save important data structures. In the main computation loop, the application does computation work and creates new versions of global arrays. It periodically calls the ABFR() function to check the application data. Four operators are implemented inside application. The ABFR runtime invokes the error check. And if an error is detected, it triggers inverse propagation operator to find PRCs and use diagnosis and recovery operators to focus recovery on PRCs. The runtime manages versions for operators. In addition, when parallel diagnosis and recovery mode are enabled, the runtime collects available computing resources and distributes workload to parallelize these procedures, achieving load balance.

3.4 Runtime: Efficient, Parallel Recovery

ABFR provides a clear separation between application knowledge and the details of the underlying systems. Application designers implement four ABFR operators using application knowledge. The ABFR runtime triggers and composes operators to achieve latent error recovery. In addition, given available resources, ABFR can further improve performance by parallelizing diagnosis and recovery. A key element of ABFR's version management support is providing global-naming, enabling easy work redistribution for load balance [35].

3.4.1 Basic Operator Scheduling

Basic Function. Application designers set the frequency of error check (e.g. error latency bound). The ABFR() function is then invoked periodically and calls error check operator to examine application data. When an error is detected, the ABFR runtime triggers

inverse propagation, diagnosis and recovery operators appropriately to identify corrupted data, diagnose potential root causes and correct all affected states.

Efficiency. The application-defined ABFR operators are triggered by the ABFR runtime. However the operators are sometimes not necessary to be invoked. For instance, in simulation of a certain problem, the inverse propagation may identify all data are potentially corrupted due to the communication pattern. Hence it is not necessary to perform. Another case is that the diagnosis cannot significantly reduce the number of PRCs. Therefore performing it renders no benefit. ABFR runtime measures the cost of operators and makes the tradeoff to optimize efficiencies.

3.4.2 Sequential and Parallel Orchestration.

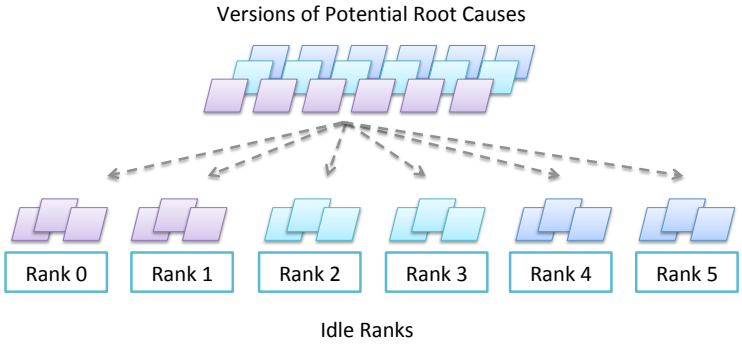


Figure 3.7: Parallel Orchestration: potential root causes are distributed for efficient concurrent diagnosis and recovery.

While we have presented ABFR as if the diagnosis and recovery operators are sequentially ordered and not overlapped with the application, this is generally not so. In many cases, application semantics allow not only parallelism within operators, but also across operators, and between the operators and the application. ABFR not only enables these forms of parallelism, but also supports their efficient exploitation with parallel scheduling and load balance. This activity is depicted in Figure 3.7. We describe several examples.

- **Parallel Operators** In parallel applications, it's often natural to specify operators

as parallel computations. However, because ABFR focuses on analysis, mechanisms for specifying tasks and resources are needed. ABFR uses versions, and PRC's within them as the implicit basis for parallel tasks.

- **Parallelism across Operators** Three ABFR operators form a natural pipeline on PRC's – inverse propagation to diagnosis to recovery. As PRC's are returned by each operator, computation for the next operator is enabled. This is a generalized flexible version of [56].
- **Load Balance** With focused recovery, the native application work and data distribution typically produces poor load balance for the three key ABFR operators. Exploiting the global view provided by ABFR's version system, the ABFR runtime redistributes tasks and data for good load balance.
- **Scheduling of Operator Tasks** ABFR runtime collects available resources, specified by the application setting the variable `ABFR_PRC_RANK` for each idle process. ABFR divides these compute resources to match the number of versions within the error latency bound and assigns the recovery tasks for one version to each partition.

The parallelism within operators and load balance are critical to reducing recovery latency in all our applications. Effective scheduling is essential in translating reductions in recovery cost to reductions in recovery latency.

ABFR enables parallel diagnosis and recovery, distributing versions of PRCs to available idle tasks and performing operators concurrently. In addition, the ABFR runtime implements overlapped, local recovery with[56], but extends them in scope and with sophisticated diagnosis. Specifically, ABFR enables only the processes whose data is affected by errors to participate in the recovery process, and other processes to continue computation (overlapping recovery, subject to application data dependencies). By bounding error scope, ABFR saves CPU throughput, reducing recovery cost. Furthermore, overlapping recovery and computation can reduce runtime overhead significantly, enabling tolerance of high error rates.

3.4.3 Version Management and Global View

Versioning of application data is accomplished by using GVR (see Chapter 2 and [1]). GVR is easy to use. Application designers allocate global arrays, and the ABFR runtime triggers versioning. The GVR system provides global view (global naming) and efficient versioning. Global naming enables diagnosis and recovery to reload only needed data (partial) from the relevant versions. This targeted partial materialization reduces I/O cost. Global naming allows the ABFR runtime to redistribute work and data for load balance in recovery. This is essential to achieving good recovery latency as discussed in Chapter 5.

The example in Figure 3.7 has three intermediate versions for diagnosis and recovery. The potential root causes are just partial data. It is often the case that only partial computation or a few processes are effected by the error, leaving the rest idle during recovery. Consequentially, the recovery work is not balanced. Instead of reloading all the versions, ABFR uses GVR to materialize only the suspicious data – potential root causes, significantly reducing I/O cost. With the version management and global view support, ABFR exploits the idle resources and redistribute the recovery workload to achieve load balance and accelerate the recovery procedure.

3.4.4 Summary of ABFR Approach

We propose a general framework for designing efficient and scalable latent error resilience – Application-based Focused Recovery (ABFR). ABFR exploits application knowledge to confine recovery scope and focus recovery on potentially corrupted data. Therefore it can significantly reduce recovery cost. The two key ideas of ABFR are (1) define the required application knowledge for latent error recovery and encapsulate them in four operators; (2) provide a powerful runtime to manage the recovery procedure using four operators. ABFR provides a range of flexible choices for application designers to express their knowledge in four operators. And meanwhile ABFR eliminates their concern of the underlying system details by managing and orchestrating the complex recovery procedure in runtime.

CHAPTER 4

APPLICATION STUDIES

To demonstrate its generality, we apply ABFR to three diverse scientific computation archetypes – stencil, N-Body tree, and Monte Carlo particle transport computations. These computations are widely used in materials science, fusion energy and chemistry (top 3 science categories on United States Department Of Energy's (DOE) National Energy Research Scientific Computing Center (NERSC) [10] supercomputer). The archetypes are a challenging set, varying widely in communication pattern (fixed-regular to dynamic-irregular), data structure (dense array, tree, discrete), and as we will show, application recovery strategy. For each, we first describe the exemplary computation structure, then describe one design for ABFR (four operators) that is used for experiments. Because the ABFR ideas focus on three operators – inverse propagate, diagnosis, and recovery – we describe them first. To illustrate the flexibility ABFR provides, we also discuss a few variants of ABFR operators. For each example, we provide an analytical performance model. The model can be used to determine the optimal error check frequency and versioning frequency.

4.1 Stencil Computations

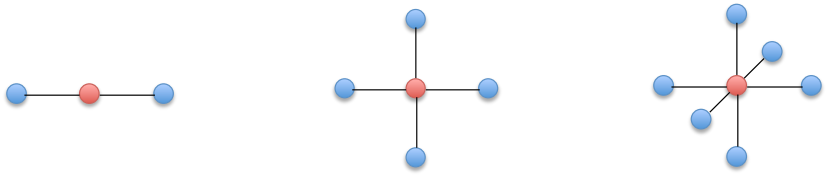
4.1.1 Stencil Archetype Overview

Stencils are a class of iterative kernels that update array elements in a fixed stencil pattern. Stencils are contiguous, so stencil applications are characterized by localized and spatial neighbor communication. Stencil-based kernels are the core of a significant set of scientific applications [39, 48], (e.g. cosmology, combustion and image processing). Stencil codes iterate on an array:

Figure 4.1 illustrates three small stencils; the computation dataflow in one iteration propagates errors only locally (to neighbors). Scaled up, in a large parallel system, each MPI rank (process) has a local computation, and then a ghost-halo exchange [73].

Algorithm 2 Stencil Computation Algorithm Structure

for K timesteps **do**
    Update each array element using neighbors in a fixed pattern
    Exchange the new value with neighbors
end for



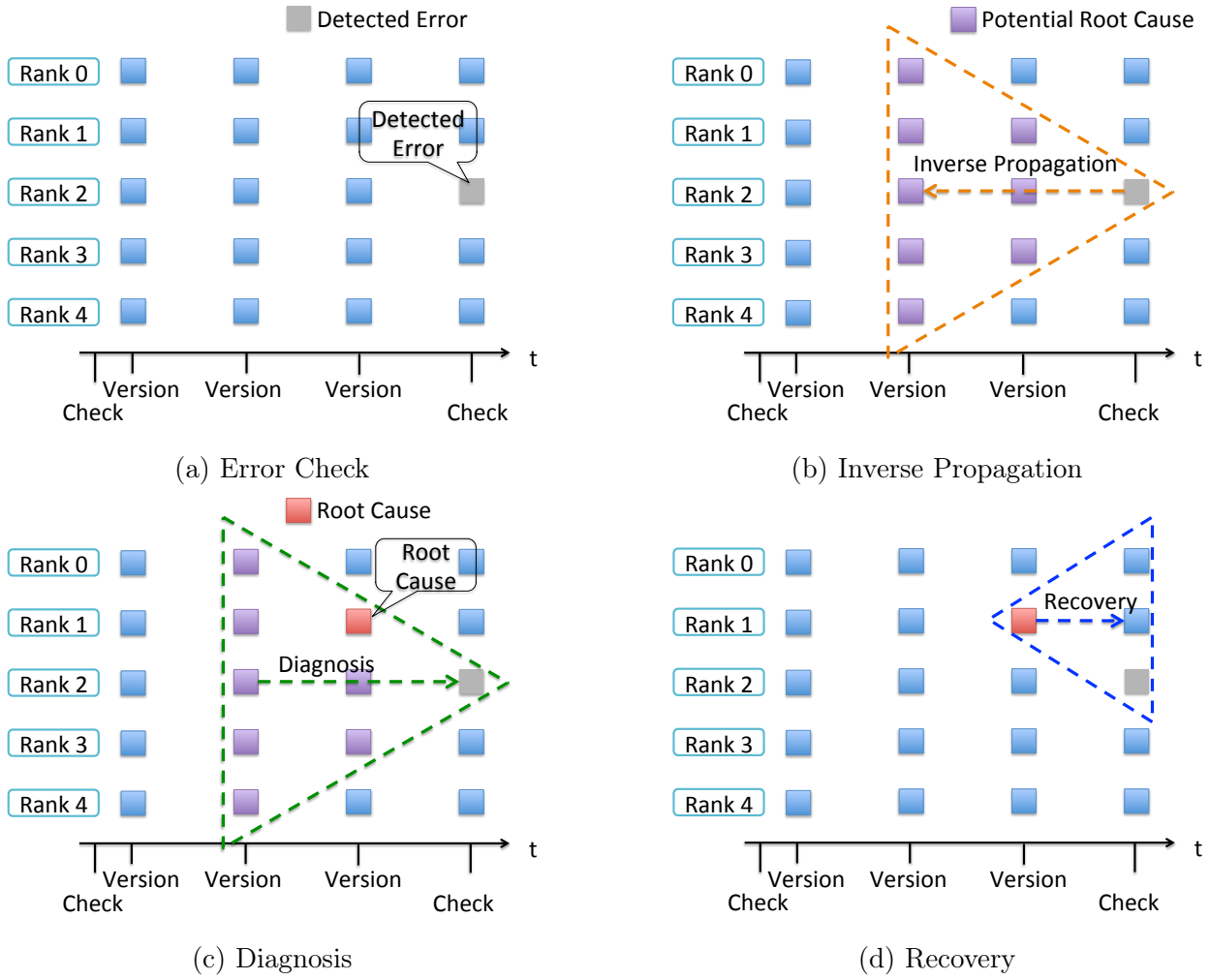
(a) 3-point 1D (b) 5-point 2D (c) 7-point 3D

Figure 4.1: Stencil patterns: an error propagates to direct neighbors (blue) in a timestep.

4.1.2 ABFR Operators for Stencil

Inverse Propagation. Stencil computations have fixed, predictable dataflow. We invert it (and larger-scale halo exchange) to accurately bound worst-case error propagation. For ABFR, our operator combines the location and latency bound from the error check to identify all data that could have contributed (blocks in the triangle in Figure 4.2b). We use this operator for experiments in Chapter 5. Alternatives for ABFR inverse propagation include a range of coarser approximations across networks of neighbor processes based on latency and error manifestation location. In the limit, these approximations would produce CR-style recovery, involving the entire application for the latency bound.

Diagnosis. In general, the inverse propagation method for stencils identifies a set of PRC ranks and simulation time for those ranks. The ABFR runtime captures versions of application state periodically, so to winnow PRCs, each of these subcomputations can be recomputed (from previous versions) and checked (against the next version). If the values match, the computation is validated, and the PRC can be pruned. Note that the GVR system provides parallel, named access to the entire <data, version> space, so it is easy to specify the recomputation to check the entire set of PRCs. Further, parallel access to the versions enables the entire checking (recomputation and comparison) process to be executed



(a) Error Check

(b) Inverse Propagation

(c) Diagnosis

(d) Recovery

Figure 4.2: Applying ABFR in a 3-point 1D Stencil.

in arbitrary order. An aggressive approach would be to compute them all in parallel, filling the idle resources typically available during error recovery. Or, if some parts of the space were more likely PRC's (error value, machine reliability), they could be prioritized.

Recovery. With a reduced set of PRCs identified, the recovery operator must recover the application state to the current time. Here we use recomputation, starting from the PRC time, and computing forward. The application specified recovery operation must have data dependences, ensuring the "cone" of application that needs to be recomputed is executed in the appropriate order (see Figure 4.2d). Note that extension to multiple PRC recovery is straightforward.

Error Check. For stencil applications the simulation smoothness can be used to detect latent errors (silent data corruption) [23]. We use a simple threshold method to detect latent errors. The location that exceeds the threshold is the error. Others employ a range of stencil semantics [22, 87, 43], including (i) one point within a range compared to its direct neighbors, and (ii) average or total heat conservation, including fluxes.

We present a stencil example on five MPI ranks in Figure 4.2. Each box is the data of one rank; ranks exchange data with neighbors each timestep, using the incoming data for the next step. When an error is detected, inverse propagation identifies all of the potential root causes (PRCs) (purple boxes). Diagnosis reduces PRCs, leaving only one viable (red box). Recovering the red box and its neighbors produces the corrected application.

4.1.3 ABFR Analytical Model for Stencil

Suppose the stencil works on M elements, each updated every timestep. Every D timesteps, an error detector is invoked to examine the state of M elements. Therefore the error latency bound is D timesteps. Then, a version of the state is stored. For ABFR, additional versions of data are created every V timesteps between two error detections. In order to simplify the model, we make the following assumptions:

- Errors occur randomly in space and time.
- Only a single error occurs between two error detections.
- Only a single manifestation of the error is detected.

Note that these assumptions are commonly used to model CR. The implications are as follows: since no other error can occur between two checks, only one recovery is needed (no error strikes during recovery). Although these assumptions cover most cases in practice, it is possible to extend the analysis to handle additional errors (see Chapter 7 for a discussion).

If an error is detected, we first identify the potential root causes based on stencil pattern. Let $step(j)$ be the number of additional elements that got corrupted after j timesteps. This typically depends on the dimension of the grid, and the number of neighbors involved in

Variable	Definitions
M	Application size (number of elements $\times$ element size)
m	Box Size (number of elements in one box $\times$ element size)
n	Number of boxes assigned to one process
p	Number of processes in computation
t	Time to advance one element by one timestep
d	Time to run the detector on one element
s	Time to store one element (versioning)
r	Time to reload one element
c	Time to compare one element with a previous version
D	Detection interval, Error Latency Bound
V	Versioning interval
α	Ratio of versioning interval to detection interval, $V = \alpha D$
B	Number of versions between two detections, $B = \frac{D}{V} = \frac{1}{\alpha}$
λ	Error rate
λM	System error rate
$(1 - e^{-\lambda M})$	Probability of having an error in one second
E	Expected cost of completing computation of D timesteps
Rec	Recovery cost: the amount of work required to recover
T	Expected runtime of completing computation of D timesteps
$RecLat$	Recovery latency: runtime critical path for recovery

Table 4.1: Summary of Main Notations for Stencil ABFR Analytical Model

the computation for one timestep. We define $root(i)$ as the number of potential root causes i timesteps ago and $AllRoot$ as the total number of potential root causes over the past D timesteps as follows:

$$root(i) = 1 + \sum_{j=1}^{i} step(j),\ AllRoot = \sum_{i=0}^{D-1} root(i)\ .$$

	1D	2D	3D
$step(i)$	2	$4i$	$4i^2 + 2$
$root(i)$	$2i + 1$	$2i^2 + 2i + 1$	$1 + \frac{4}{3}i^3 + 2i^2 + \frac{8}{3}i$
$AllRoot$	D^2	$\frac{2}{3}D^3 + \frac{1}{3}D$	$\frac{1}{3}D^4 + \frac{2}{3}D^2$

Table 4.2: Expressions for *step*, *root*, and *AllRoot* functions for 1, 2 and 3 dimensional grids, assuming an element interacts only with its direct neighbors.

Table 4.2 shows the expressions for *step*, *root* and *AllRoot* for 1D, 2D, and 3D stencils. Diagnosis is done by recomputing elements from the last correct version, which was D timesteps ago, and by comparing the results against intermediate versions. If the recomputed data differs from the version, then the error occurred between the last two versions. Note that with a version at every step, we can narrow the root cause of an error to a single point. Suppose the error occurred j timesteps ago, then the time required for diagnosis is the time to reload the last correct version, $r \cdot root(D)$ and the time to recompute, reload and check $(t + r + c)$ each element against the version from iteration $D - 1$ to j as illustrated in Figure 4.2c:

$$diag(i) = r \cdot root(D) + (t + r + c) \sum_{j=i}^{D-1} root(j) \ .$$

Once potential root causes are pruned, recovery is done by recomputing the reduced set of potential root causes and affected data , as illustrated in Figure 4.2d:

$$recomp(i) = (t + s) \sum_{j=1}^{i} root(j) \ .$$

As discussed in Chapter 3, ABFR allows overlapping recovery. In that case, the recovery cost (work needed) is the critical metric. If recovery cannot be overlapped, then recovery latency (parallel time) is appropriate. We model both of these for 2D stencils. We refer the reader to the extended version of this paper [15] for the analysis of 1D and 3D stencils.

Recovery Cost Let $\mathbb{E}_{ABFR}$ denote the total cost (amount of work due to computation, detection, versioning and recovery, counted in CPU time) of the ABFR approach, as a function of error rate λ (errors per second per byte) and detection interval D. In this section, we compare it with the classical CR (Checkpoint/Restart) approach, denoted by $\mathbb{E}_{CR}$.

Program execution is divided into equal-size segments of D timesteps. The time needed

to complete one segment with p processes is $\frac{DtM}{p}$, and the total CPU time on computation is DtM. Similarly, we spend a total of dM time on detection and BsM time on versioning, where B is the number of versions taken between two detections. For CR, we use $B = 1$, as CR creates a version every D timesteps. Then, we assume that errors occur following an exponential distribution, and the probability of having an error during the execution of one segment is denoted by $1 - e^{-\lambda M \frac{DtM}{p}}$, where λM is the application error rate. Therefore, we can write $\mathbb{E}_{CR}$ and $\mathbb{E}_{ABFR}$ as functions of D and λM as follows:

$$\mathbb{E}_{CR} = DtM + dM + sM + \left(1 - e^{-\lambda M \frac{DtM}{p}}\right) Rec_{CR} \ , \tag{4.1}$$

$$\mathbb{E}_{ABFR} = DtM + dM + BsM + \left(1 - e^{-\lambda M \frac{DtM}{p}}\right) Rec_{ABFR} \ . \tag{4.2}$$

The main difference between both approaches lies in recovery cost. Recovery of CR includes reloading data and full recomputation, while ABFR includes diagnosis cost, different data reloading, and reduced recomputation cost. For CR, we have:

$$Rec_{CR} = rM + DtM \ . \tag{4.3}$$

For ABFR, let $B = \frac{D}{V}$ denote the number of versions taken between two detections. We number versions backwards, from $j = 0$ (timestep 0) up to $j = B - 1$ (timestep $(B-1)V$). The last checked version (timestep D) has been versioned too ($j = B$). We introduce the notation $A(j)$, which is the total number of potential root causes between two versioned timesteps jV and $(j+1)V$, excluding $(j+1)V$ but including jV:

$$A(j) = \sum_{k=jV}^{(j+1)V-1} root(k) \ .$$

Therefore, $\frac{A(j)}{AllRoot}$ denotes the probability that the error occurred between version j and

$j+1$, and we can write:

$$Rec_{ABFR} = \sum_{j=0}^{B-1} \frac{A(j)}{AllRoot} \left(diag(j) + recomp(j)\right) \ .$$

The diagnosis is done by recomputing all potential root causes from timesteps $D-1$ up to version j, that is timestep jV. In addition, we need to pay $(r+c)root(kV)$ for every version k that passed the diagnosis test, that is from version $B-1$ to j included. Therefore, we can write:

$$diag(j) = r \cdot root(D) + t \sum_{k=jV}^{D-1} root(k) + (r+c) \sum_{k=j}^{B-1} root(kV) \ .$$

Because we may have gaps in-between versions, we do not know the exact location of the root cause of the error. Therefore, we recompute starting from version $j+1$ instead of j. We must recompute all potential affected elements from timestep $(j+1)V-1$ to 0. At timestep $(j+1)V-1$, there are $root((j+1)V-1)$ potential root causes elements to recompute. At every timestep, the number of elements to recompute increases by $step(j)$, so that there are a total of $root(2(j+1)V)$ elements to recompute at timestep 0. Therefore, we can write:

$$recomp(j) = t \sum_{k=(j+1)V-1}^{2(j+1)V} root(k) + s \sum_{k=j+1}^{2(j+1)} root(kV) \ .$$

Simplifying the above equation, and keeping higher order terms only (w.r.t. D), we obtain the following recovery cost as a function of the detection interval D:

$$\begin{gathered} Rec_{ABFR} = \frac{8}{15} t(\alpha^5 - 5\alpha^3 + 9\alpha + 5)D^3 + O(D^2), \\ \text{where } \alpha = \frac{1}{B}. \end{gathered} \tag{4.4}$$

Recovery Cost Comparison The dominant cost in recovery is recomputation. It is $O(DM)$ for CR in Equation 4.3 and $O(D^3)$ for ABFR in Equation 4.4. Suppose the number

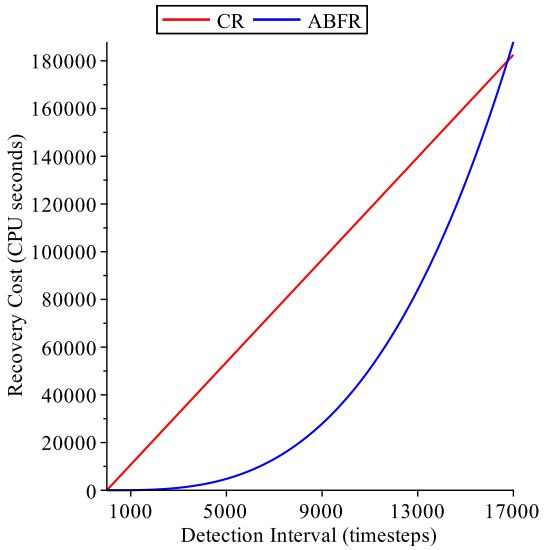


Figure 4.3: Recovery Cost vs. Detection Interval ($M = 32768^2, t = 10^{-8}, d = 100t, r = 10^{-9}, s = 10^{-8}, \alpha = \frac{1}{4}$)

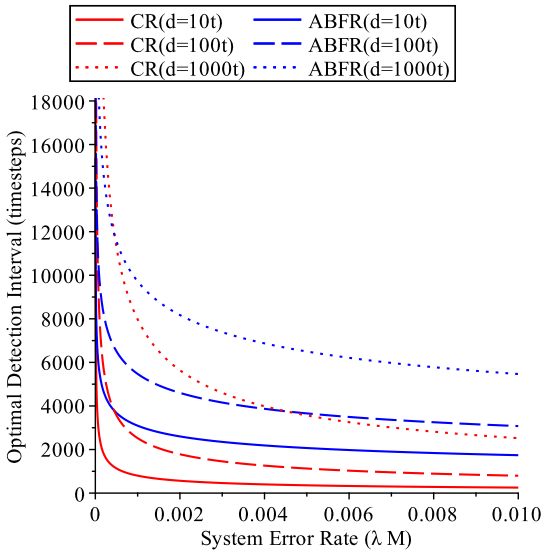


Figure 4.4: Optimal Detection Interval vs. Error Rate ($M = 32768^2, p = 4096, t = 10^{-8}, r = 10^{-9}, s = 10^{-8}, \alpha = \frac{1}{4}$)

of elements in one dimension of stencil is U, we have $M = U$, $M = U^2$ and $M = U^3$ for 1D, 2D, and 3D stencil respectively. Since CR always recomputes all the data, the corresponding recomputation cost is $O(DU)$, $O(DU^2)$ and $O(DU^3)$. In contrast, ABFR only needs to recompute a small fraction of the M elements. The corresponding recomputation cost is $O(D^2)$, $O(D^3)$ and $O(D^4)$ respectively (see [15]). Note that the detection interval D (or error latency) is much smaller than the number of elements in one dimension U.

We plot the recovery cost of CR and ABFR as a function of detection interval (error latency) in Figure 4.3 (note that CR creates 1 version during D timesteps, while ABFR creates B versions. The plot uses $B = \frac{1}{\alpha} = 4$). We observe that CR grows linearly with detection interval. While ABFR increases slowly for less than 9,000 and outperforms CR for error latencies up to 17,000 timesteps. This range of 1,000 to 17,000 timesteps corresponds to 3 seconds to about 1 minute. After that, most data are corrupted, hence ABFR cannot further improve the performance by bounding error impact.

Let $H = \frac{\mathbb{E}}{DtM}$ denote the expected overhead with respect to the computation cost without

errors. Using Taylor series to approximate $\left(1 - e^{-\lambda M \frac{DtM}{p}}\right)$ to $\lambda M \frac{DtM}{p}$ (up to first-order terms), we obtain:

$$
\begin{aligned}
H_{CR} &= 1 + \frac{d+s}{Dt} + \frac{\lambda M}{p}(rM + DtM), \\
H_{ABFR} &= 1 + \frac{b}{D} + \frac{\lambda M}{p} a D^3, \\
\text{where } a &= \frac{8}{15} t(\alpha^5 - 5\alpha^3 + 9\alpha + 5) \text{ and } b = \frac{\alpha d + s}{\alpha t} \ .
\end{aligned}
\tag{4.5}
$$

Optimal Detection Interval Minimizing the overhead, we derive the following optimal detection interval for Checkpoint-Restart and ABFR:

$$
D^*_{CR} = \sqrt{\frac{(d+s)p}{\lambda M^2 t^2}} \ , \text{ and } D^*_{ABFR} = \sqrt[4]{\frac{bp}{3a\lambda M}} \ . \tag{4.6}
$$

Empirical studies of petascale systems have shown MTBF's of three hours at deployment [41], and allowing for the greater scale of exascale systems [89, 29], future HPC system MTBFs have been projected as low as 20 minutes [54]. To explore possibilities for a broad range of future systems (including cloud), we consider system error rates (errors/second) ranging from 0 (infinite MTBF) to 0.01 (1 minute MTBF). We assume the application runs on the entire system, setting λM to the system error rate.

We plot the optimal detection interval as a function of the error rate λM in Figure 4.4. We observe that as error rate increases, the optimal detection interval of CR drops faster than ABFR for varied error detector cost, indicating CR demands more frequent error detection in high error rate environments. So, here the goal is to be lazy in error detection checking, because deep application-semantics are assumed to be expensive. Higher numbers for optimal

detection interval are good. Plugging D^* back into H, we derive that

$$H^*_{CR} = 1 + 2M\sqrt{\frac{(d+s)}{p}}\sqrt{\lambda} + rM^2\lambda \ , \tag{4.7}$$

$$H^*_{ABFR} = 1 + \frac{4}{3}\sqrt[4]{\frac{3ab^3\lambda M}{p}} \ . \tag{4.8}$$

We plot the overhead as a function of error rate, when using the optimal detection interval, in Figure 4.5. With growing error rates, CR incurs high overhead. In contrast, ABFR significantly reduces overhead and performs stably even for high error rates.

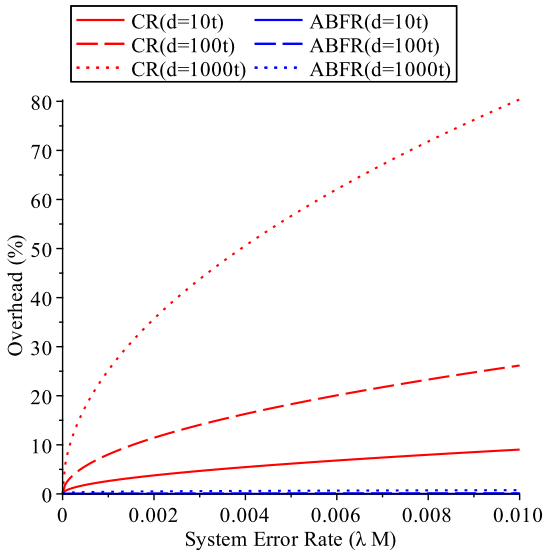


Figure 4.5: Overhead vs. Error Rate Using Optimal Detection Interval ($M = 32768^2, p = 4096, t = 10^{-8}, r = 10^{-9}, s = 10^{-8}, \alpha = \frac{1}{4}$)

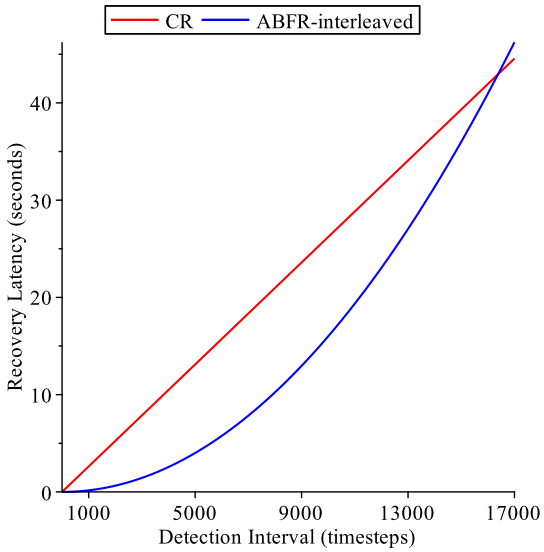


Figure 4.6: Recovery Latency vs. Detection Interval ($M = 32768^2, m = 65536, p = 4096, n = 4, t = 10^{-8}, d = 100t, r = 10^{-9}, s = 10^{-8}, \alpha = \frac{1}{4}$)

Recovery Latency We model recovery latency (parallel execution runtime). Large-scale simulations overly decompose a grid into boxes, enabling parallelism and load balance. As in Figure 4.7, each process is assigned a set of boxes; each of which is associated with a halo of ghost cells. The square grid of $\sqrt{M} \times \sqrt{M}$ elements is partitioned into square boxes of size $\sqrt{m} \times \sqrt{m}$. We have $\frac{M}{m}$ boxes mapped on to p processes.

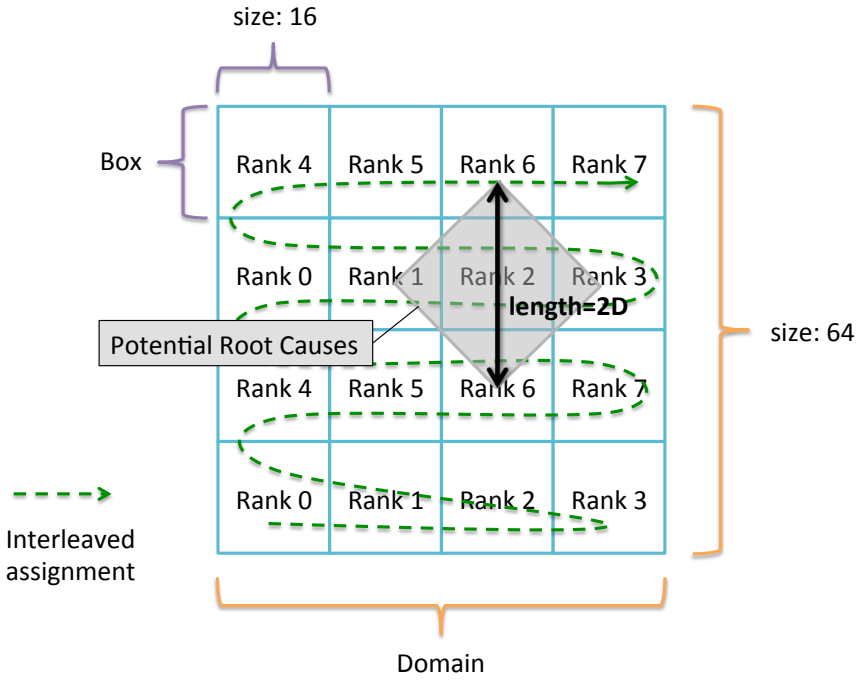


Figure 4.7: Interleaved Domain Decomposition

Recovery latency, $RecLat$, is determined by the process with the most work. For CR, we assume perfect load balance; each process has n boxes, so $npm = M$. Thus $RecLat_{CR}$ reloads n boxes and recomputes them for D timesteps:

$$RecLat_{CR} = n(rm + Dtm) \ . \tag{4.9}$$

For ABFR, recovery latency is determined by the process with the most corrupted boxes. For simplicity, we recompute entire box even it is partially corrupted in ABFR. In an ideal case, the actual corrupted boxes are owned by processes uniformly, making the number of corrupted boxes of each process, equal to $n_{ideal} = \frac{root(D)}{mp} = \frac{2D^2}{mp} + O(D)$. For the interleaved mapping (see Figure 4.7), there are $\sqrt{M/m}$ boxes in one row, so the vertical distance between two boxes assigned to the same rank is $\frac{p}{\sqrt{M/m}}$ (box). The length $2D$ is the range of error spread. The slowest process would have $n_{inter} = \frac{2D}{\sqrt{m}} / \frac{p}{\sqrt{M/m}} = \frac{2D\sqrt{M}}{mp}$ corrupted boxes.

Then, for an error at step j, we have:

$$diag(j) = rm + t \sum_{k=jV}^{D-1} m + (r+c) \sum_{k=j}^{B-1} m,$$
$$recomp(j) = t \sum_{k=0}^{(j+1)V} m + s \sum_{k=0}^{j+1} m \ .$$

To compute the recovery latency Rec_{box} per box, we proceed as before:

$$\begin{aligned} Rec_{box} &= \sum_{j=0}^{B-1} \frac{A(j)}{AllRoot} \left(diag(j) + recomp(j)\right) \\ &= tm\alpha D + o(D). \end{aligned}$$

Multiplying Rec_{box} by the corresponding number of boxes in the ideal and interleaved scenarios, we obtain

$$RecLat_{ideal} = \frac{2t\alpha}{p} D^3 + O(D^2), \tag{4.10}$$
$$RecLat_{inter} = \frac{2t\alpha\sqrt{M}}{p} D^2 + O(D) \ . \tag{4.11}$$

Comparing Equations (4.9) and (4.10), we conclude that as long as the latency is not long enough to infect all assigned boxes of one process, ABFR would produce better performance. We plot $RecLat_{CR}$ and $RecLat_{inter}$ as a function of detection interval in Figure 4.6. Similar as in Figure 4.3, CR increases linearly with detection interval. And ABFR outperforms CR for the detection interval from 0 to 17,000 timesteps. But the gap between their recovery latencies is smaller compared with that in recovery cost. The gap between recovery latencies mainly depends on the difference in the number of boxes that the slowest process needs to work on. Therefore ABFR is at most $n = 4$ times better in the plot configuration.

Optimal Detection Interval. We derive the expected runtime of CR and ABFR to

successfully compute D timesteps.

$$T_{CR} = Dnmt + dnm + snm + (1 - e^{-\lambda MDnmt})RecLat_{CR}$$
$$T_{ABFR} = Dnmt + dnm + Bsnm + (1 - e^{-\lambda MDnmt})RecLat_{ABFR}$$

The overhead $H = \frac{T}{Dnmt}$ of CR and ABFR are given by

$$H_{CR} = 1 + \frac{d+s}{Dt} + \lambda Mn(rm + Dtm),$$
$$H_{ideal} = 1 + \frac{\alpha d + s}{\alpha Dt} + \lambda M \frac{2t\alpha}{p} D^3,$$
$$H_{inter} = 1 + \frac{\alpha d + s}{\alpha Dt} + \lambda M \frac{2t\alpha\sqrt{M}}{p} D^2 \ .$$

Minimizing the overhead, we derive the optimal detection interval for CR, ideal ABFR and interleaved ABFR respectively:

$$D^*_{CR} = \sqrt{\frac{(d+s)p}{\lambda M^2 t^2}}, D^*_{ideal} = \sqrt[4]{\frac{(\alpha d + s)p}{6\alpha^2 t^2 \lambda M}}, D^*_{inter} = \sqrt[3]{\frac{(\alpha d + s)p}{4\alpha^2 \lambda M^{\frac{3}{2}} t^2}} \ .$$

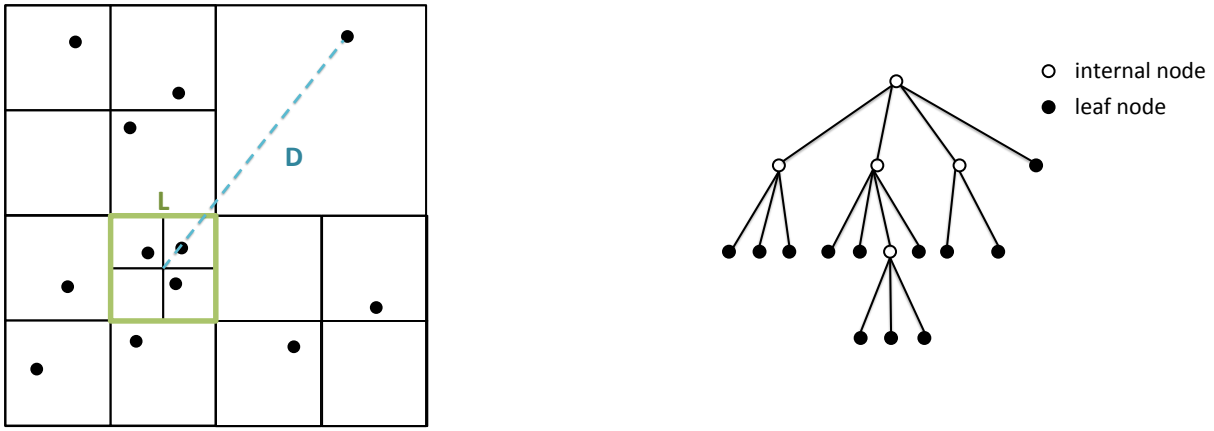
The optimal interval D^*_{CR} of CR is the same as in Equation (4.6). The optimal interval for ideal-ABFR is $D^*_{ideal} = \Theta(\lambda^{\frac{-1}{4}})$, the same order of magnitude as D^*_{ABFR}, the optimal value of Equation (4.6) for the recovery cost. D^*_{inter} is different due to imbalanced recovery.

4.2 N-Body Tree Computations

4.2.1 N-Body Archetype Overview

The N-Body computations model the dynamics of a set of bodies (or particles), subject to force at a distance (e.g. gravity, charge attraction, etc.). N-Body simulations are a fundamental tool in astrophysics, molecular dynamics, and even materials simulations, and underlie an entire class of scientific applications [13, 14].

Modern computational methods for N-Body simulations employ tree methods to achieve efficiency, reducing complexity from $\mathcal{O}(N^2)$ to $\mathcal{O}(N \log N)$. These gains are achieved by summarizing the effect of groups of particles with careful consideration of how force contributions decrease with distance [47, 93, 16, 58]. We select Barnes-Hut [19] as an exemplar of this class of tree codes.



(a) 2D Spatial Domains

(b) Quad-Tree Representation

Figure 4.8: Barnes-Hut Structures: N-Body Tree Code

Barnes-Hut (BH) subdivides 2D space repeatedly in a quadtree structure (octree for 3D) with four daughters for each node on down to single particles (see a 2D example in Figure 4.8). Each internal tree node is a "virtual particle" summary of the mass and position of all of its children. Forces are obtained for each particle by walking the tree from root on down, one-level at a time. At each level, BH applies an error criteria to decide whether to use the summary or "open" the node, and explicitly consider the next level of descendants. BH decides to open a node if $l/D > \theta$, where l is the region-length (one side) of the region represented by the node, and D is the distance from the particle to the node's center-of-mass. θ is typically 0.3 to 0.8. If $l/D < \theta$, then the subtree is approximated by the node's "virtual particle" summary.

Tree reconstruction can be expensive and is unnecessary if the particle configuration has changed little. Applications using N-Body tree methods balance tree construction time

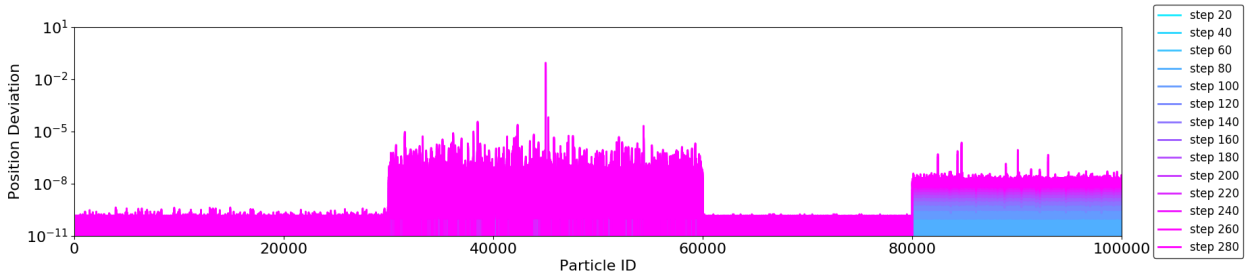


Figure 4.9: Error Propagation, two galaxy collision simulation with 100k particles and 32 processes. A bit-flip error is injected into bit 47 of one particle's velocity in step 20. Y-axis shows the position deviation of each particle compared to error free run for step 20 to 280. The error corruption is concentrated in subtrees for certain latencies while the rest particles have small deviations.

against force computation and particle movement time for best efficiency [70]. We adopt a dynamic update scheme where tree nodes can be updated without reconstructing the entire tree at every timestep [91]. Algorithm 3 shows the pseudocode of N-Body tree computations.

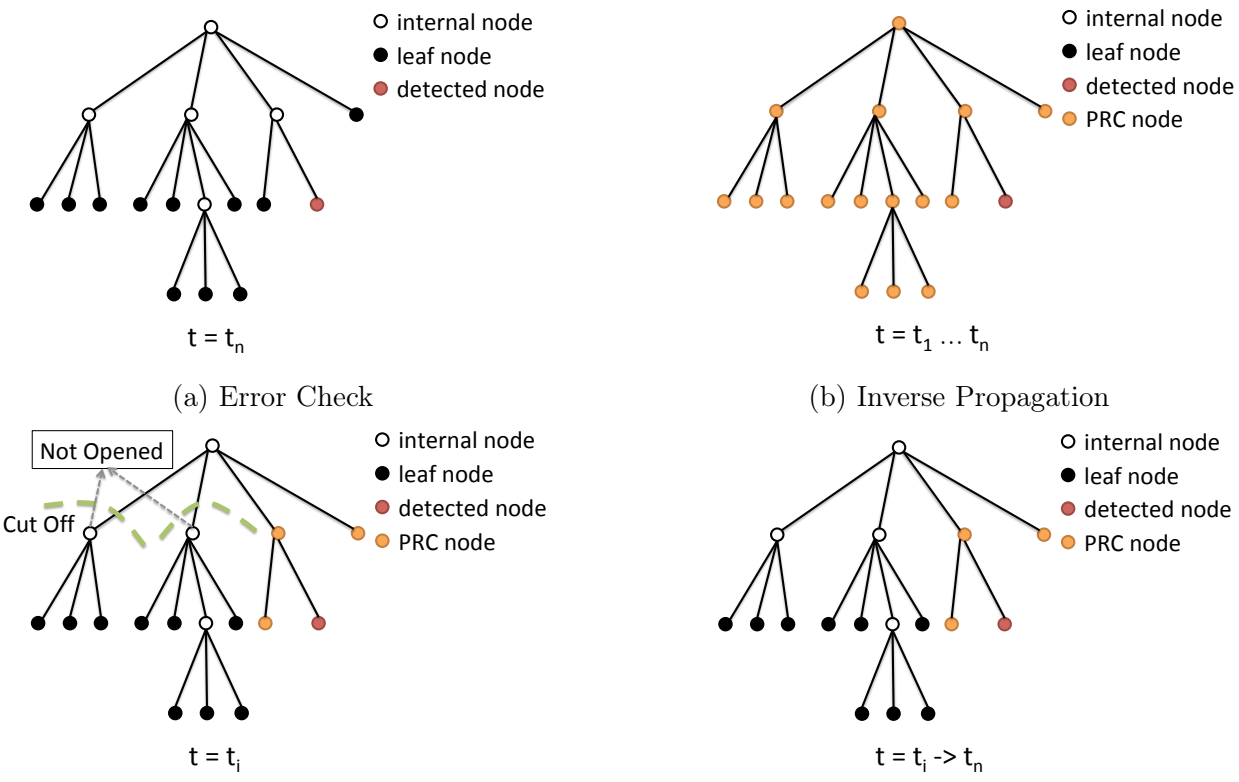
Algorithm 3 N-Body Tree Computation Algorithm Structure

```
for K timesteps do
    Construct Barnes-Hut tree, update internal nodes.
    for k_local timesteps do
        for N particles do
            Walk tree (Open or Not) and compute forces
        end for
    end for
end for
```

N-Body tree computations are challenging for latent-error resilience. A particle error can spread through the entire simulation in a single tree update (e.g. all tree nodes contaminated). In large simulations, the key result is often the evolution of the particle distribution, not the behavior of individual particles. Thus simulation results with small deviations may be valid, and only unacceptable when an appreciable fraction of the particles have diverged [61]. Experiments injecting bit-flip errors into varied particles and bit positions show clustered propagation tied to tree structure. For example, errors injected into particle velocity, and low-order mantissa bits (bit 47), give opportunities for focused recovery (see Figure 4.9).

Thus our ABFR approach detects when errors are clustered, and focuses diagnosis to reduce error recovery cost.

4.2.2 ABFR Operators for N-Body



(a) Error Check

(b) Inverse Propagation

(c) Diagnosis

(d) Recovery

Figure 4.10: Applying ABFR in a 2D N-Body Tree Computation.

Inverse Propagation. Given an error manifestation – erroneous particle – the tree structure and walks since the prior error check define the particles (and node summaries) that contributed to the erroneous particle, so they are all PRCs. At worst, all nodes were opened, so the PRCs include all tree nodes (including internal nodes and leaf nodes) as shown in Figure 4.10b. ABFR versions the N-Body tree after each reconstruction, so each of these PRCs can be examined after a latent error has been detected.

Diagnosis. Given the PRCs, the diagnosis operator considers each versioned N-Body tree. For each tree, it simulates a tree walk, using the opening criterion as a cut-off to

prune subtrees (internal nodes and their children) from PRCs (see Figure 4.10c). Nodes not opened are unlikely to be PRC's and are pruned. For nodes that were opened, recompute and compare the positions of particles to previously saved results. If the differences are within a given threshold, the corresponding nodes are pruned from PRCs.

Recovery. Given the reduced set of PRCs, recompute PRCs and their downstream paths for each step until current timestep. Because many restored trees were pruned, the required computation for this set of PRCs is much less than a full recomputation.

Error Check. System energy change is a widely-used error measure in N-Body computations [61, 97], checking physical energy conservation. Since the geometric structure of tree sometimes evolves slowly in time, a large jump (out of a cell) for single particles indicates an error. The error check returns one erroneous particle.

We show an example in Figure 4.10. An error is detected in a particle, and all nodes are initially PRCs in inverse propagation. For each versioned tree, using the BH opening criteria prunes two subtrees from PRCs. Recomputing the remaining PRCs and comparing with the versions finds the true root cause at step i. Recovery recomputes the root causes and opened neighbors.

4.2.3 ABFR Analytical Model for N-Body Tree

We consider a N-Body simulation of N particles. The height of resulted Barnes-Hut tree is H. Particle are updated every timestep with cost c per particle. Every V timesteps, the tree is reconstructed. Before the update, a version of all particle data is created, with cost v per particle. The error check is invoked every L timesteps, namely, the error latency bound.

We define opening rate – for a single particle force computation, the ratio of internal nodes that are opened at certain level of the tree. The opening rate depends on the mass distribution of the simulation and the position of the particle being simulated. Suppose the average opening rate at level h of the tree is $f(h)$.

Versioning. Versioning is performed every V timesteps. The cost is v per particle. The

Definitions	
H	Height of tree
N	Number of particles
Error Rate	
λ	Errors per second per particle
Time	
c	Time to compute one leaf
d	Time to detect errors on one leaf
v	Time to version one leaf
r	Time to reload data of one leaf
Tree-wise	
T_c	Time to compute the tree without errors
T_d	Time for detection the tree without errors
T_v	Time for versioning the tree without errors
Frequency	
L	Latency Bound (Detection interval) timesteps
V	Versioning interval (timesteps)
Functions	
$f(h)$	Opening rate at tree height h

Table 4.3: Summary of Main Notations for N-Body Tree ABFR Analytical Model.

total cost of saving N particles is $N \cdot v$.

Detection. Error check is performed every L timesteps. The total cost of checking N particles is $N \cdot d$, where d is the time to apply the error check on single particle.

Expected Cost of Checkpoint-Restart Let $\mathbb{E}(T_{CR})$ denote the expected total cost of Checkpoint-Restart approach including computing, versioning, detection and recovery.

$$\mathbb{E}(T_{CR}) = T_c + N \cdot (d + v) + (1 - e^{-\lambda T_c})T_c \ .$$

Expected Cost of ABFR Let $\mathbb{E}(T_{ABFR})$ denote the expected cost of ABFR. The recovery cost of ABFR includes inverse propagation, diagnosis and recomputation.

Inverse Propagation: by our analysis, all particles are potential root causes. The actual inverse propagation procedure do not need to be performed. Thus the cost is 0.

Diagnosis: there are $\frac{L}{V}$ versions in the period. For each period, we restore the tree structure and prune all nodes that are not opened at tree level h. The number of reduced set of PRCs is $f(h) \cdot N$. The diagnosis cost include recomputing the reduced set of PRCs and comparing with stored versions, $(r + c) \cdot f(h) \cdot N$. The expected actual error latency is half of the interval. Therefore the expected diagnosis cost is given by

$$\mathbb{E}(T_{diag}) = (r + c) \cdot f(h) \cdot N \cdot \frac{L}{2V}$$

Recomputation: recompute PRCs and their downstream neighbors.

$$\mathbb{E}(T_{recomp}) = \sum_{i=1}^{a} (f(h) \cdot N)^i \cdot c, \text{ where } a = \frac{L}{2V}$$

$$\begin{aligned}\mathbb{E}(T_{ABFR}) &= T_c + N \cdot (d + v) + (1 - e^{-\lambda T_c})(T_{diag} + T_{recomp}) \\ &= T_c + N \cdot (d + v) + (1 - e^{-\lambda T_c})((r + c) \cdot f(h) \cdot \frac{L}{2V} + \sum_{i=1}^{\frac{L}{2V}} (f(h) \cdot N)^i \cdot c)\end{aligned}$$

Optimizing $\mathbb{E}(T_{ABFR})$ derives the optimal error check interval and versioning interval for N-Body tree computations. The results highly depend on the opening rate of the tree walk. Prior to the actual run, the opening rate can be sampled through trial runs, and used to tune the error check interval and versioning interval.

4.3 Monte Carlo Particle Transport

4.3.1 Monte Carlo Archetype Overview

Particle transport is the study of motions and interactions of neutrons or photons with materials. It is widely-used to model nuclear processes, radiography, medical physics, computer

graphics, accelerator target design, and reactor design. Monte Carlo (**MC**) is used to solve particle transport problems [21, 26, 69]. The MC trials are samples of the particle transport process. Particles transport within a predefined geometry space, and encounter probabilistic events (i.e. scattering, absorption, and fission) with corresponding parameters (i.e. distance, energy, angle, etc.) that are determined by known probability distributions. Computation results are accumulated scores from the trials and resulting events into sums called *tallies*. Thus the communication pattern is of largely independent computations, combining their results in the tally array. With enough particles, tallies (normalized appropriately) converge to stable result value. That is, their statistical error generally decreases as the number of trials increases. Sophisticated Monte Carlo computations use the statistical error estimates for current results to "target" the trials, doing intelligent sampling to speed convergence. MC particle transport is described in Algorithm 4.

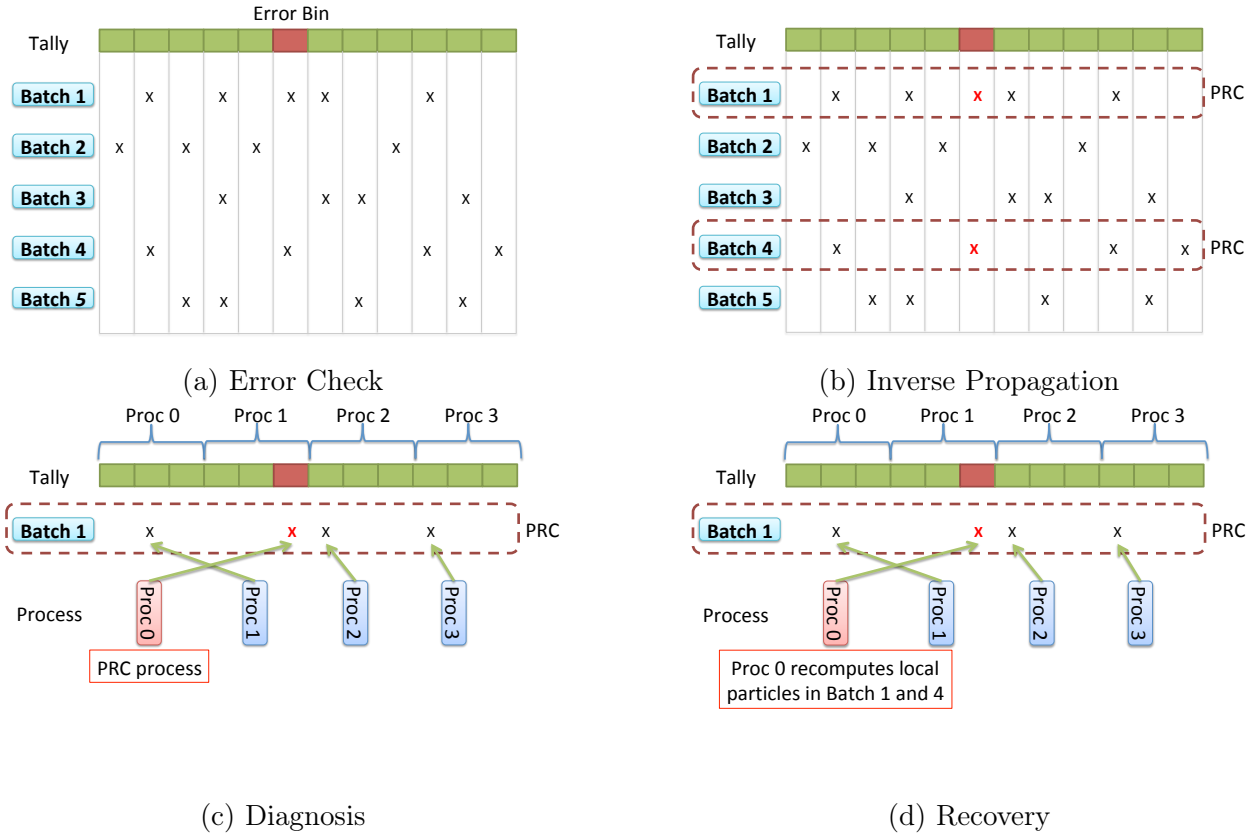
Tally data, the results, is partitioned over the MPI ranks (processes). The tally data size depends on the number of physical quantities and physical regions studied, and in a realistic reactor simulations, can reach terabytes.

Algorithm 4 Monte Carlo Particle Transport Algorithm Structure

```
for K batches do
    for N particles do
        Simulate the motion/movement of the particle
        if Event satisfies filter criteria then
            for all Scoring functions do
                Calculate score
                Accumulate score to tally array
            end for
        end if
    end for
end for
```

4.3.2 ABFR Operators for Monte Carlo

Inverse Propagation. MC particle-transport computations have dynamic data flow – the tallies affected by a trial depends on statistical events (integral to the scientific model)



(a) Error Check

(b) Inverse Propagation

(c) Diagnosis

(d) Recovery

Figure 4.11: Applying ABFR in a Monte Carlo Particle Transportation Computation. 'X' mark indicates particles in that batch scored to the corresponding bin.

in the simulation. To deal with this, the ABFR inverse propagation operator requires an application change, we add a data structure to track the history of dependence – a bit-vector map of the contribution of each batch to each tally (see Figure 4.11b). Using this bit-vector map and the error manifestation location (a tally bin), inverse propagation inverts the map to identify all the batches that are PRCs.

Diagnosis. Tally arrays are distributed across the MC computation processes. Each process contributes to a subset of the tallies. To exploit this, the ABFR diagnosis operator requires an application change, adding a bit-vector map used to record communication dependencies between processes for tally accumulation as shown in Figure 4.11c. With this bit-vector communication map, the diagnosis operator can eliminate PRCs within the previously suspected batches. The particles of the remaining processes are the winnowed PRCs,

reducing recovery scope from entire batches to a few processes. An alternative is to simply avoid the cost of the bit-vector map for communication, and use the entire batches for recovery. This would increase recovery cost, but if errors are sufficiently infrequent might be advantageous.

Recovery. The recovery operator removes the contribution of the remaining PRCs then recomputes those trials. The versioned tallies are used to remove the contribution of PRC processes, and then the trials are recomputed and added to the tally results.

Error Check. As tallies converge, the MC computation is able to estimate a statistical error bound. If the change within a batch exceeds the error bound, it is considered a manifestation of a latent error. The location of the manifestation is the tally bin whose change exceeds the bound, and the value is the tally.

We illustrate MC particle transport in Figure 4.11. There are five batches in simulation. The inverse propagation utilizes the bit-vector map and identifies Batch 1 and 4 as contributing to the detected erroneous tally in process 1. Diagnosis with the communication map finds that only process 0 communicated with process 1 in these two batches. Consequently, only the particles of process 0 remain as PRCs. Process 0 removes its contribution and recomputes and accumulates the results.

4.3.3 ABFR Analytical Model for Monte Carlo Particle Transport

Expected Cost of Checkpoint-Restart Let $\mathbb{E}(T_{CR})$ denote the expected total cost of Checkpoint-Restart approach. It includes computing, versioning, detection and recovery.

$$\mathbb{E}(T_{CR}) = T_c + B \cdot d + v + (1 - e^{-\lambda T_c})T_c.$$

Expected Cost of ABFR **Inverse propagation:** use the bitvector map to identify all batches that contributed to the detected erroneous tally bin as PRCs. The computational cost is negligible. We define a batch contribution factor f_{batch}, the possibility of particles in

N	Number of particles per batch
B	Number of batches
S	Tally size
L	Latency Bound (Detection interval), batches
V	Versioning interval (batches)
c	Time to compute one particle
d	Time to detect one batch
v	Time to version one batch
Functions	
f_{batch}	Contribution factor
$f_{particle}$	Contribution factor

Table 4.4: Summary of Main Notations for MC Particle Transport ABFR Analytical Model.

one batch scoring to a specific tally bin. The number of PRC batches is given by

$$B \cdot f_{batch}.$$

Diagnosis: for each PRC batch, identify group of particles that scored to the detected erroneous tally bin using the map. We define a particle contribution factor $f_{particle}$, the possibility of a particle scoring to a specific tally bin. The total number of PRC particles derived is

$$B \cdot f_{batch} \cdot N \cdot f_{particle}.$$

Recovery: recompute PRC particles and accumulate results. The cost is given by

$$T_{rcmp} = B \cdot f_{batch} \cdot N \cdot f_{particle} \cdot c.$$

The total cost of ABFR is

$$\begin{aligned}\mathbb{E}(T_{ABFR}) &= T_c + B \cdot d + v + (1 - e^{-\lambda T_c}) T_{rcmp} \\ &= T_c + B \cdot d + v + (1 - e^{-\lambda T_c})(B \cdot f_{batch} \cdot N \cdot f_{particle} \cdot c).\end{aligned}$$

Optimizing $\mathbb{E}(T_{ABFR})$ derives the optimal error checking interval and versioning interval. The optimal value depends on the batch contribution factor and particle contribution factor.

4.4 Discussion and Summary

	Stencil	**N-Body Tree**	**MC Particle Transport**
Error Check	Compare the value variation of single point to a threshold.	Compare the position variation of a particle to a threshold.	Check convergence of tallies.
Inverse Propagation	Use data flow (stencil pattern) to identify PRCs.	Use tree structures to identify PRCs.	Use bit-vector map to identify PRC batches.
Diagnosis	Recompute PRCs and compare with previously saved results.	Use opening criterion to cutoff subtrees from PRCs.	Use bit-vector map to prune PRC particles.
Recovery	Recompute PRCs.	Recompute PRCs.	Recompute PRCs or forward recovery.

Table 4.5: Summary of ABFR operators for Stencil, N-Body Tree, and Monte Carlo computations

We summarize our experience of applying the ABFR approach to varied computational archetypes. We found that despite major differences in communication and parallelism structure, ABFR could be successfully applied and limited application knowledge is required.

In all three archetypes, we identified methods for *inverse propagation*. With stencils it was simple, following the regular and predictable computation data flow. For N-Body tree it was more complex, following both the tree data structures, and tree-walking computation structures. For Monte Carlo, there was not even enough extant structure to follow, so inverse propagation required first the addition of a tracking data structure (bit-vector maps) to enable inverse propagation. For diagnosis, we were able to find techniques to winnow PRCs for each of the three application archetypes. It was most difficult for N-Body tree computations, but using knowledge of application error propagation enabled us to find effective techniques. The N-Body tree algorithms intrinsically have error management and control, which can

be easily adopted by ABFR diagnosis operator to prune PRCs. For all three types, there were several viable options for diagnosis. Interestingly, the GVR versioning system enable parallel diagnosis in many cases, an unexpected benefit. Recovery was straightforwardly implemented as recomputation, but approximations based on deeper application knowledge are also possible. For example, forward recovery [64, 68] and approximate recovery [36] could also be used. Table 4.5 compares the design of four operators for three computations.

Three examples show that limited application knowledge is required for designing ABFR operators. Some of them is already managed by application designers, intrinsically part of the application. Such examples include the data flow used in stencil inverse propagation operator, the tree-walking algorithm and error control used in N-Body tree inverse propagation and diagnosis operator. ABFR can easily frame these knowledge in four operators without much additional effort.

The set of variants discussed in operator design show that flexible choices of application knowledge are available. Application designers can easily experiment with these different strategies by encapsulating them in four operators. ABFR enables latent error resilience design within a well defined framework.

CHAPTER 5

PERFORMANCE EVALUATION

In this Chapter, we evaluate ABFR's performance. We present the methodology, metrics, workloads and platforms used for evaluation. Three real production application codes are used. We implemented ABFR operators for each. We run experiments in significant scale (up to 4,096 processes) on world famous supercomputers. The results demonstrate ABFR is efficient and scalable.

5.1 Methodology

We evaluate ABFR for the three application archetypes, using real application codes – Chombo (stencil), Gadget2 (N-Body tree), and OpenMC (MC particle transport). In each case, we use the ABFR operator designs described in Chapter 4, and compare to a latent-error recovery scheme based on Checkpoint-Restart (CR), as described in Chapter 3.

In all cases, the applications use the GVR system to create versions, but at different frequencies. CR creates a version after each error check. One version per check is best, enabling CR to immediately identify and restore the last good checkpoint and recompute. This avoids iterative testing at the cost of more recomputation. ABFR creates additional versions between two error checks. Additional versions incur small additional runtime cost [35], and are used by ABFR to optimize latent-error recovery. The error-check intervals (i.e. error latency bound) and versioning intervals affect overall application performance. Optimal intervals depend both on error rates and operator cost in a fashion similar to checkpoint period optimization [98, 38]. In other work, we built an analytical models for ABFR that can determine the optimal intervals [50, 31].

We study recovery performance, varying error latency bound. The bounds are determined by checking cost and error rates. We run experiments for a set of error latency bounds, corresponding to a range of error rates. For each, we perform three trials and average results.

For stencil study, we further validate the analytical performance model and examine the cost of ABFR using optimal error checking interval derived by the model. For Monte Carlo particle transport, we vary both the scale and error latency bounds. By exploring ABFR's performance under the circumstance of weak-scaling and strong-scaling, we demonstrate the scalability of ABFR.

5.1.1 Metrics

We use two metrics – *recovery cost* and *recovery latency*.

- **Recovery cost.** The total work (CPU time) consumed across all nodes of the parallel system to complete recovery.
- **Recovery latency.** The critical path runtime for application recovery.

As discussed in Chapter 3, ABFR allows overlapping recovery. The recovery cost (work needed) measure is most relevant if recovery can be overlapped with other application computation (or other use of the system). If recovery cannot be overlapped, then recovery latency (parallel time) is the most relevant measure.

5.1.2 Platforms

Experiments were conducted on two platforms.

- **Midway2.** Shared compute cluster at the University of Chicago [5], 382 nodes, linked by non-blocking FDR/EDR Infiniband. Nodes: 28-core, 2.4Ghz Intel Broadwell with 64 GB memory.
- **Edison.** DOE supercomputer [4], Cray XC30 system at National Energy Research Scientific Computing Center (NERSC), 5,586 nodes, linked by Cray Aries Dragonfly. Node: dual 12-core, 2.4Ghz Intel IvyBridge with 64GB memory.

5.1.3 Applications, Settings, and Workloads

Chombo Heat Equation. Chombo [37] is a library that implements block-structured adaptive mesh refinement technique. We use Chombo 2D heat equation codes; these codes solve a parabolic partial differential equation for heat distribution in a region over time. It is a 5-point 2D stencil program and uses an interleaved domain decomposition method for load balance.

The Chombo 2D code is run for a domain of 10^9 elements that is divided into 16,384 boxes. The error latency bound is varied from 1,000 to 13,000 timesteps, producing the potentially corrupted data sizes from 0.2% to 32% of the total data set. ABFR creates 4 versions of the domain between error checks. As the latency bound is increased, the interval between versions also increases. CR creates a version after each error check. This computation is run on 4,096 processes (MPI ranks), spread over 342 compute nodes on the Edison supercomputer.

Gadget2. Gadget2 [90] is a parallel MPI code for cosmological N-Body simulation. It computes gravitational forces using a hierarchical tree algorithm. Gadget2 has been used to address a wide array of interesting astrophysics problems, ranging from colliding galaxies to structure formation at the origin of the universe.

We run a Gadget2 simulation of galaxy collision where two disk galaxies run into each other, leading to a merger. Each galaxy consists of a stellar disk (200,000), and a massive and extended dark matter halo (300,000). The simulation has 1 million particles in total with Gadget2 updating the N-Body tree every 10 timesteps and ABFR versioning after each tree construction. CR creates a version after each error check. We vary error latency from 100 to 300 timesteps. Gadget2 is run on Midway2 with 128 processes across 32 nodes.

OpenMC. OpenMC [80] is a production Monte Carlo particle transport simulation code, developed by Computational Reactor Physics Group at the Massachusetts Institute of Technology. It is used by DOE CESAR co-design center to explore scalable nuclear reactor modeling.

OpenMC is run on the Monte Carlo Performance Benchmark [62], simulating 128,000 particles. The physical parameter – neutron production rate – is tallied over a $289 \times 289 \times 100$ mesh comprising 8,352,100 tallies (or tally bins) that are a total of 128 Megabytes of data [81]. Each batch simulates 128,000 particles, and the error latency bound is varied from 10 to 30 batches. Both ABFR and CR create versions of the tally array after each batch. OpenMC is run with 128 processes on 32 nodes on Midway2.

5.2 Chombo Results (Stencil)

5.2.1 Recovery Cost

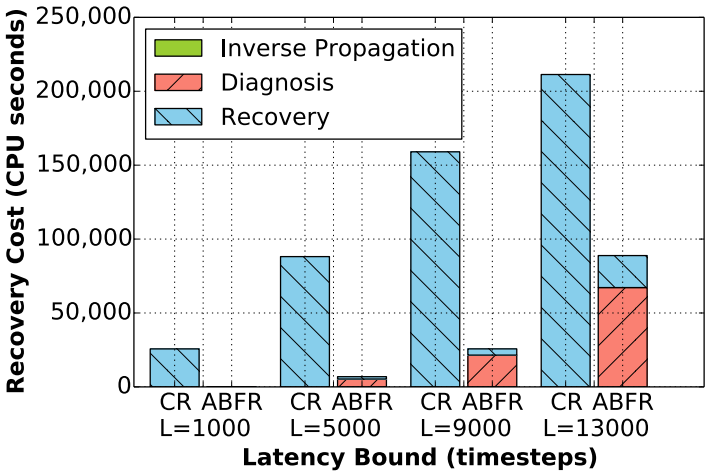


Figure 5.1: Stencil Recovery Cost: ABFR vs. CR, various error latency bounds.

In Figure 5.1, we compare the recovery cost for Chombo for a range of error latency bounds (1,000 to 13,000 timesteps). As expected, recovery cost for CR grows linearly with the error latency bound. ABFR achieves a recovery cost that is 367x lower at 1,000 timesteps (70 vs. 25,700 CPU seconds), and grows slowly. The gap between them increases steadily but the ratio decreases. Even at 13,000 timesteps, ABFR has 2.4x lower recovery cost than CR. Figure 5.2 shows the breakdown of ABFR cost. The cost of ABFR diagnosis increases with error latency bound, because more PRCs need to be tested. The number of PRCs

grows cubically as a function of error latency [50]. Note that diagnosis for ABFR Chombo is highly effective, pruning 99% PRCs and reducing recovery work. ABFR recovery work also increases with error latency bound, and at 13,000 timesteps is 10% of the CR recovery cost. This reflects the essential recomputation work given the data corruption from the latent error. Inverse propagation is small, independent of error latency bound.

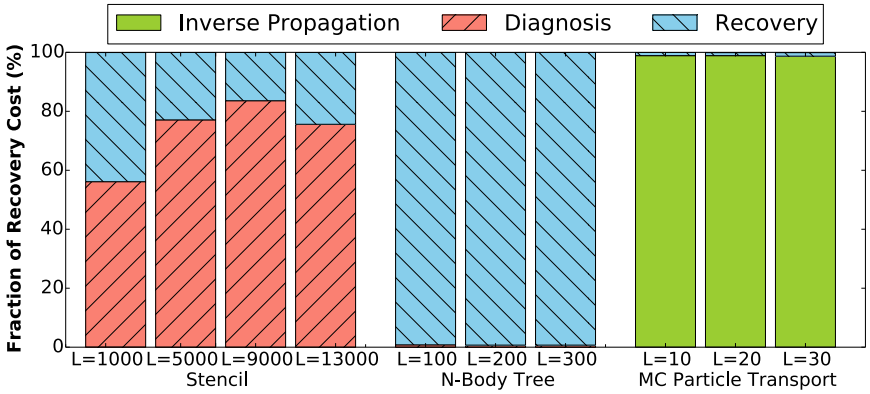


Figure 5.2: Operators as fraction of overall ABFR recovery cost

	Latency Bound (timesteps)	Inverse Propagation	Diagnosis	Recovery
Recovery Cost	1,000	0.04%	56.1%	43.8%
	5,000	0.0%	77.1%	22.9%
	9,000	0.0%	83.6%	16.4%
	13,000	0.0%	75.6%	24.4%
Recovery Latency	1,000	0.0%	56.1%	43.9%
	5,000	0.0%	62.8%	37.2%
	9,000	0.0%	62.9%	37.1%
	13,000	0.0%	54.1%	45.9%

Table 5.1: Operators as fraction of overall ABFR recovery: Stencil

5.2.2 Recovery Latency

Figure 5.3 compares the recovery latency for a range of error latency bounds. ABFR reduces the recovery latency by about 4x at 1,000 timesteps and 2.2x at 13,000 timesteps. Both parallelizing and load-balancing of diagnosis enabled by ABFR is critical to the reduction

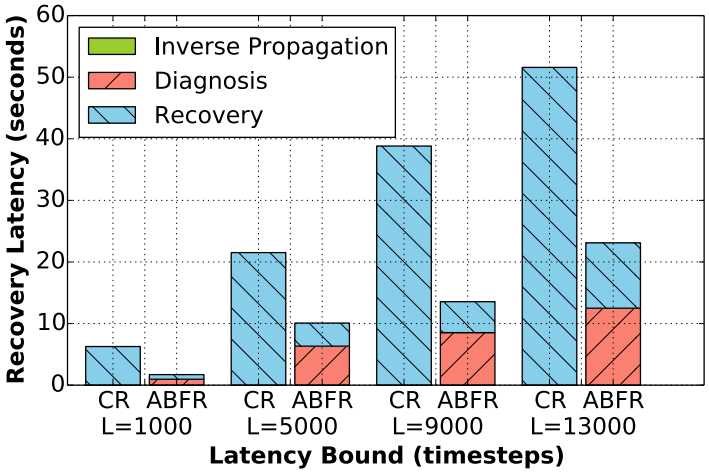


Figure 5.3: Stencil Recovery Latency: ABFR vs. CR, various error latency bounds.

in latency. For 1,000 timesteps, only 41 boxes are PRCs, and for 13,000 the number of boxes identified as PRC's has grown to 5,380 with a load imbalance of 4:1. This imbalance produces the faster diagnosis recovery latency growth that reduced ABFR benefit with our first diagnosis operator implementation to only 1.08x at 13,000 timesteps. Table 5.1 presents the breakdown of ABFR operator cost. The cost of inverse propagation is negligible. The dominant cost is consumed by diagnosis and recovery, because these two operators perform the main computation work to prune PRCs and correct the states.

Taking advantage of ABFR flexibility, we improved the diagnosis operator, using the reload of data from version system as an opportunity to rebalance, and reducing load imbalance to 3:2. To give a sense of the importance of these optimizations, in Figure 5.4, we compare our initial naive approach (*sequential*) that does diagnosis for each version in sequence, and uses the existing application data decomposition to distribute work. Comparison to our parallel, load balanced version (*parallel*) shows a 1.5x to 3.1x improvement. This is critical to ABFR doubling its improvement to 2.2x overall lower recovery latency at error-latency bounds of 13,000 timesteps.

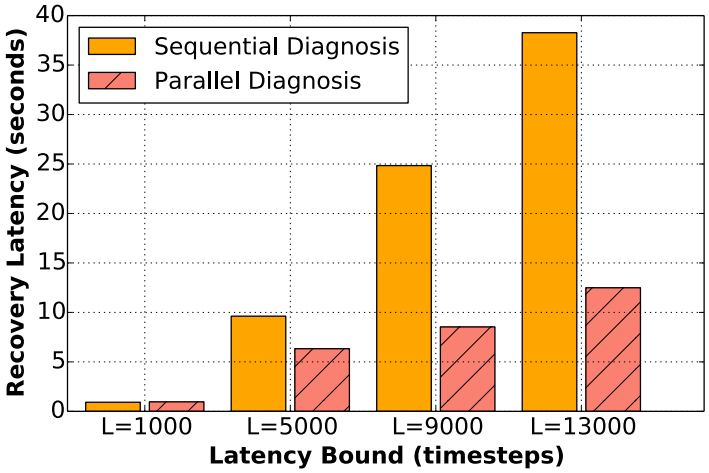


Figure 5.4: Stencil: Parallel Diagnosis vs. Sequential Diagnosis

Number of ranks	4096
Domain size	10^9 (32768x32768)
Number of boxes	16384 (128x128)
Box size	65536 (256x256)
#Box per process	4

Table 5.2: Experiment Configurations for Analytical Model Validation

5.2.3 Analytical Model Validation

Experiment Design

We explore the performance of CR and ABFR for varied error detection intervals and error latencies. The configuration of experiments is listed in Table 5.2. We run 4,096 ranks and solve the heat equation for a domain of 10^9 elements. With this problem size, we vary the detection interval from 1,000 timesteps to 13,000 timesteps, producing potential corrupted data fractions that range from 0.2% to 32%. ABFR always creates 4 versions, the interval between versions increases with the detection interval. For each detection interval, we sample error latencies uniformly, injecting an error in each versioning interval. We measure the performance for each error latency and calculate the average results to produce performance for the detection interval length.

All experiments were conducted on Edison, the Cray XC30 at NERSC (5576 nodes, dual 12-core Intel IvyBridge 2.4 GHz, 64GB memory). We use 4,096 ranks, typically spread over 342 nodes. The results are an average of three trials.

Results

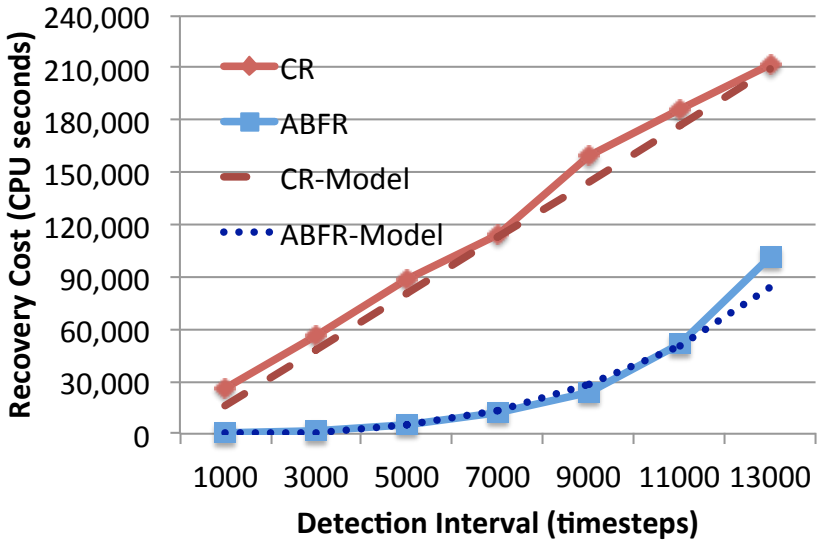


Figure 5.5: Recovery Cost vs. Detection Interval (Model plotted for experiment configuration and measured $t = 1.5 * 10^{-8} second$)

Recovery Cost Figure 5.5 plots the recovery cost for varied detection intervals (1000 to 13,000 timesteps). Recovery cost for CR grows linearly with detection interval (error latency). The recovery cost of ABFR is initially 400x lower (62 vs. 25,700 CPU seconds at 1000 timesteps), and it grows slowly. The gap between them increases steadily but the ratio decreases. Even at 13,000 timesteps, ABFR has 2x lower recovery cost. In contrast to CR, ABFR effectively focuses recovery effort on only 41 of 4096 ranks (i.e. 1%), corresponding to 13-24 nodes, depending on alignment. This reduction is the direct benefit of inverse propagation and diagnosis.

Figure 5.5 also plots the performance model (dotted and dashed lines), showing a close match (for broader comparison see Figure 4.3). As expected, ABFR cost starts lower and

grows polynomially with the detection interval.

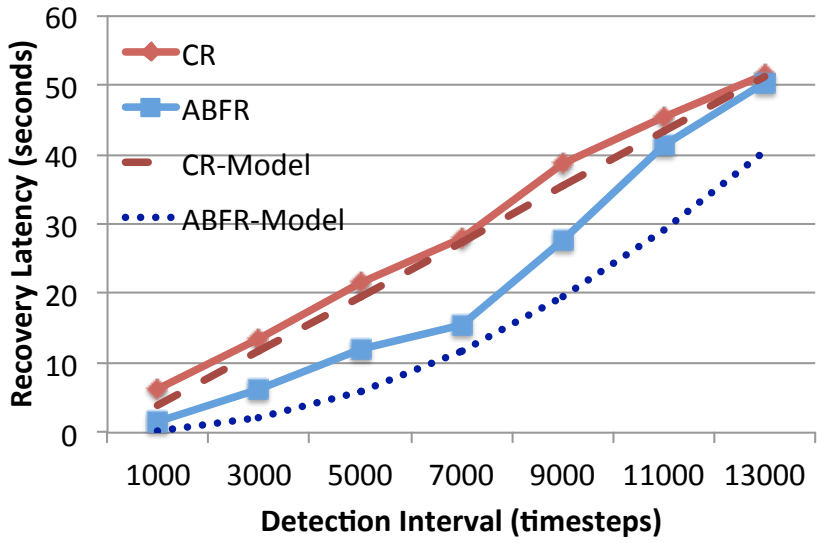


Figure 5.6: Recovery Latency vs. Detection Interval (Model plotted for experiment configuration and measured $t = 1.5 * 10^{-8} second$)

Recovery Latency Figure 5.6 compares the recovery latency with a range of detection intervals. For shorter intervals (1000 timesteps), ABFR reduces recovery latency by up to 4x. The recovery latency is determined by the slowest process. In CR, each process recomputes all 4 boxes assigned to it at every timestep. In ABFR, for 1,000 timesteps, only 41 boxes are identified potentially corrupted and processes involved in recovery work on one box at most. As detection interval increases, the error may propagate to a larger area, making it more likely that each process has more boxes to handle. At detection interval (error latency) of 13,000 timesteps, ABFR has same performance as CR.

The dotted and dash lines in Figure 5.6 are performance model results using parameter values of our experiments (see also Figure 4.6). Our experiment results have similar curves as the model. The recovery latency of CR grows almost linearly with detection intervals. While ABFR produces low recovery latencies for short detection intervals and then chases up with CR with expanding detection intervals. The measured ABFR performance are slightly worse than the model because we only keep the highest order terms in the model for simplification

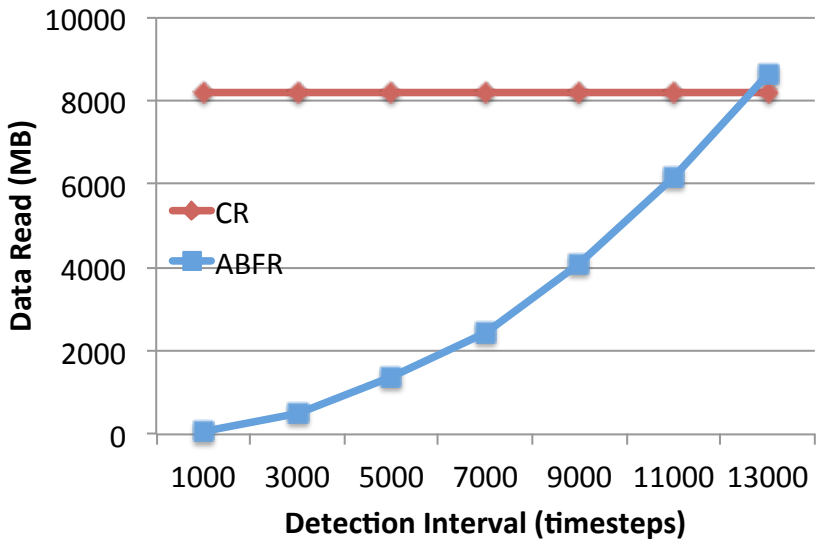


Figure 5.7: Data Read (MB) vs. Detection Interval

but omit some other costs.

Data Read (IO) An important cost for recovery is the reading of stored version data from the IO system. Figure 5.7 presents the data read versus detection intervals. In general, the data read increases with detection interval as on average the actual error latency is greater, causing ABFR to read parts of more versions. In contrast, CR always reloads the entire grid. Because ABFR intelligently bounds the error impact and loads the required data to recover all potential errors, it reduces data read by as much as 1000-fold.

5.3 Gadget2 Results (N-Body Tree)

5.3.1 Recovery Cost

Figure 5.8 shows the recovery cost for N-Body tree computation. Recovery cost of CR grows linearly with the latency bound. The recovery cost of ABFR is 8.1x, 6.4x and 7.4x lower for latency bound 100, 200 and 300 timesteps respectively. ABFR achieves better performance because diagnosis and focused recovery reduces the recovery work dramatically. Note that inverse propagation is inexpensive – all the trees and particles since the last error

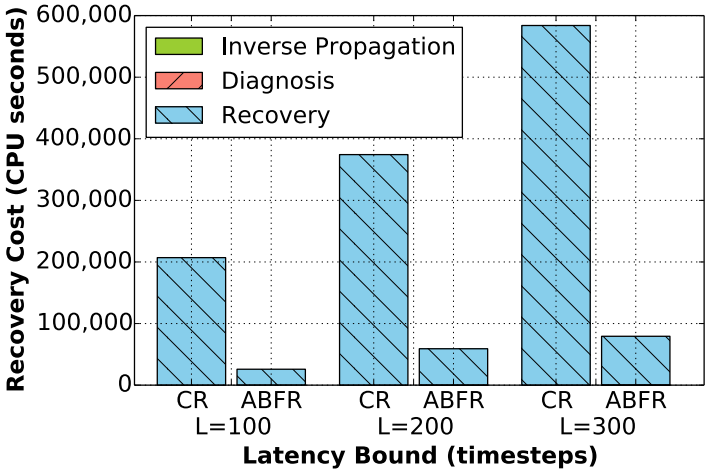


Figure 5.8: N-Body Tree Recovery Cost: ABFR vs. CR, various error latency bounds.

check are just deemed PRCs. Using the numerical cut-off (opening criteria) for diagnosis is lightweight and effective, pruning over 99% particles with less than 1% of total recovery cost (see Figure 5.2). The recovery operator recomputes the remaining PRCs for the latency, so its cost grows linearly with error latency bound.

5.3.2 Recovery Latency

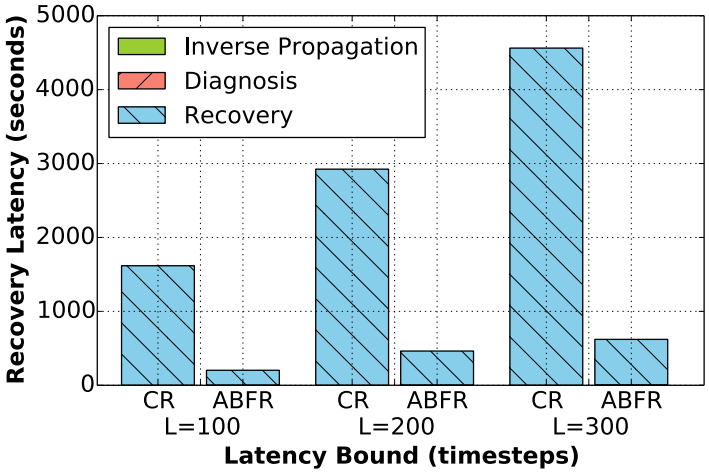


Figure 5.9: N-Body Tree Recovery Latency: ABFR vs. CR, various error latency bounds.

Figure 5.9 compares the recovery latency for varied latency bounds. ABFR reduces the

recovery latency by nearly 8x compared to CR. Because the cost of ABFR inverse propagation and diagnosis operator are so low, their structure matters little for recovery latency. The ABFR recovery operator is sequential, and its latency grows linearly with error latency bound. In general, most of the recovery cost and latency benefits can be attributed to ABFR's effective pruning that dramatically reduces the number of PRC particles, reducing recovery cost and latency.

	Latency Bound (timesteps)	Inverse Propaga-tion	Diagnosis	Recovery
Recovery Cost	100	0.0%	0.8%	99.2%
	200	0.0%	0.7%	99.3%
	300	0.0%	0.7%	99.3%
Recovery Latency	100	0.0%	1.2%	98.8%
	200	0.0%	0.9%	99.1%
	300	0.0%	1.0%	99.0%

Table 5.3: Operators as fraction of overall ABFR recovery: N-Body Tree

Table 5.3 compares the cost of ABFR operators for N-Body tree simulation. Unlike stencils, the majority of time is consumed by recovery operator in N-Body tree computation. The inverse propagation and diagnosis operators exploit algorithms and data structure properties, and are demonstrated to be lightweight.

5.4 OpenMC Results (Monte Carlo)

5.4.1 Recovery Cost

Figure 5.10 presents the recovery cost of ABFR for error latency bounds from 10 to 30 batches. While recovery cost of CR grows linearly, ABFR reduces recovery cost by over 57x. The overwhelming reason for this is the effectiveness of the bit-vector maps in filtering PRC's. The ABFR inverse propagation operator narrows from dozens of PRC batches to a few. There are only 2 PRC batches at latency 10, 4 PRC batches at latency 20, and 10 PRC batches at latency 30. The ABFR diagnosis operator uses the communication dependency

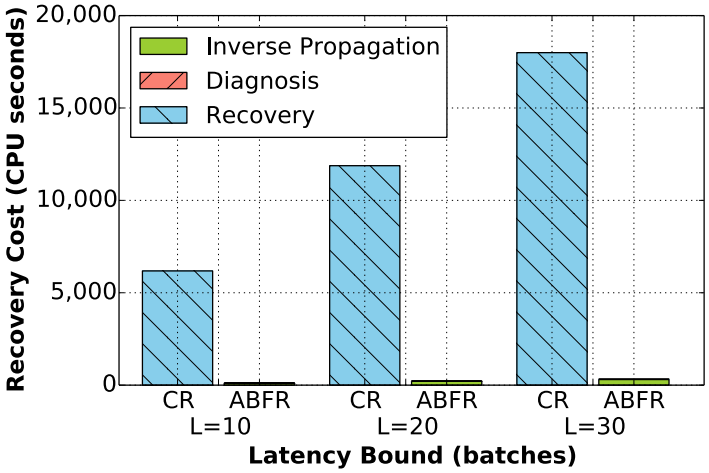


Figure 5.10: MC Particle Transport Recovery Cost: ABFR vs. CR, various error latency bounds.

map, narrowing the number of processes within these batches, finding that a single process has contributed to the detected erroneous tally. In contrast, for CR all processes and all particles are recomputed for all batches within the error latency bound.

5.4.2 Recovery Latency

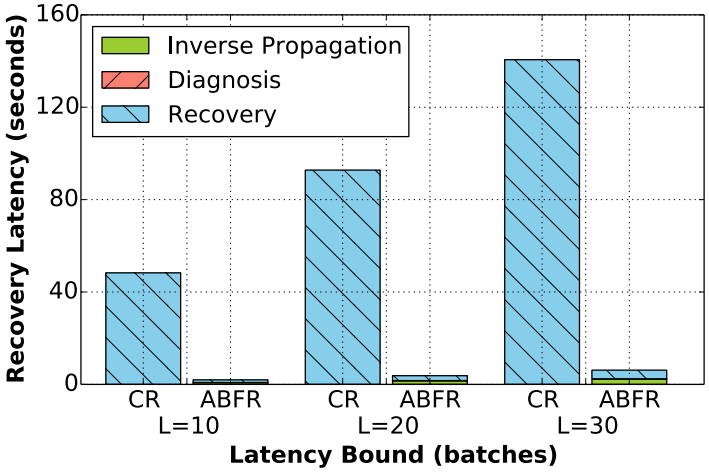


Figure 5.11: MC Particle Transport Recovery Latency: ABFR vs. CR, various error latency bounds.

For recovery latency, see Figure 5.11, the story is similar. Because the ABFR inverse

propagation and diagnosis operators for OpenMC are so effective in focusing recovery effort, there is only a small amount of work to be done. The work is focused on just one process which we do not distribute, though given this is a Monte Carlo simulation, such distribution might be possible. Despite that, recovery latency for OpenMC is improved by 24x on average (22.8x to 24.9x) by ABFR vs. CR for all error latency bounds. This improvement is slightly less than that for recovery cost.

	Latency Bound (Batches)	Inverse Propaga-tion	Diagnosis	Recovery
Recovery Cost	10	98.9%	0.0%	1.1%
	20	98.9%	0.0%	1.0%
	30	98.7%	0.0%	1.2%
Recovery Latency	10	41.8%	0.5%	58.2%
	20	43.7%	0.3%	56.3%
	30	39.4%	0.2%	60.6%

Table 5.4: Operators as fraction of overall ABFR recovery: MC Particle Transport

Table 5.4 compares the cost of each operator in Monte Carlo Particle Transport computation. Distinguished from stencil and N-Body tree's lightweight inverse propagation operator, MC Particle Transport has a relative costly inverse propagation operator. Adding the bit-vector map to track the contribution of each batch, the inverse propagation operator has more I/O cost compared to the other two studies, but effectively reduces the recovery scope.

The right set of bars in Figure 5.2 shows the cost fraction of each operator. The inverse propagation is about 98.9% of the total recovery cost. It reads the batch contribution map (1 MegaBytes per version) and identifies PRC batches. In comparison, the diagnosis cost is negligible, reading a 16-Byte communication map and pruning PRC processes. The recovery operator consumes more than 56% of the recovery latency. However since only one process is identified as PRC after diagnosis, its CPU consumption is just about 1% of total recovery cost.

5.4.3 Scaling ABFR in OpenMC

In this section, we explore the performance of ABFR when OpenMC scales up. We study both strong scaling (same amount of work with more processes) and weak scaling (same amount of work per process with more processes). We scale the number of processes to 1,024 and compare the results with 128 processes.

Strong Scaling. We run the application with same amount of work (i.e. simulating 128x1024 particles) but use 1,024 processes, that is, single process samples 128 particles in each batch. The jobs run on NERSC Edison supercomputer.

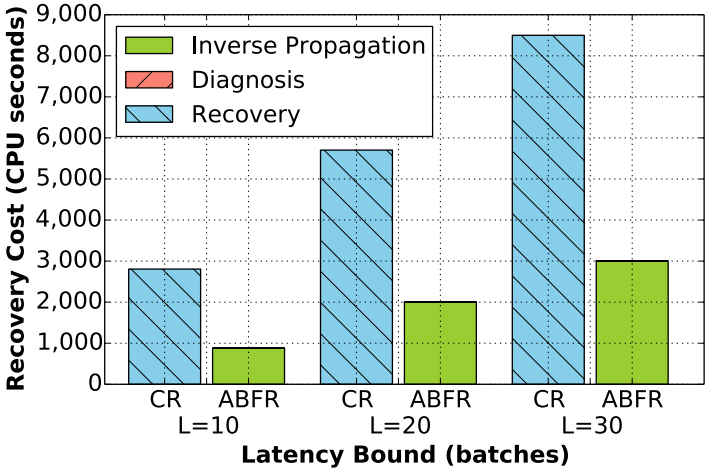


Figure 5.12: Strong scaling, MC Particle Transport Recovery Cost: ABFR vs. CR

Figure 5.12 presents the recovery cost of ABFR, and Figure 5.12 compares the recovery latency of ABFR and CR for error latency bounds from 10 to 30 batches. ABFR reduces the recovery cost by 2.8x to 3.2x and recovery latency by 2.5x to 2.65x. Using strong scaling, the total cost of CR is ideally same as smaller scale because the amount of work for recovery is same. The measured recovery cost of CR with 1024 processes is actually about 1.3x that of 128 processes. While the cost of CR grows linearly, ABFR can still focus recovery on where needed. Only two batches are identified as PRC batches in all three latency bounds. Only one out of 1024 processes contributed to the detected erroneous tally. ABFR effectively reduces the recovery scope, as shown in Figure 5.14. The amount of recomputation needed

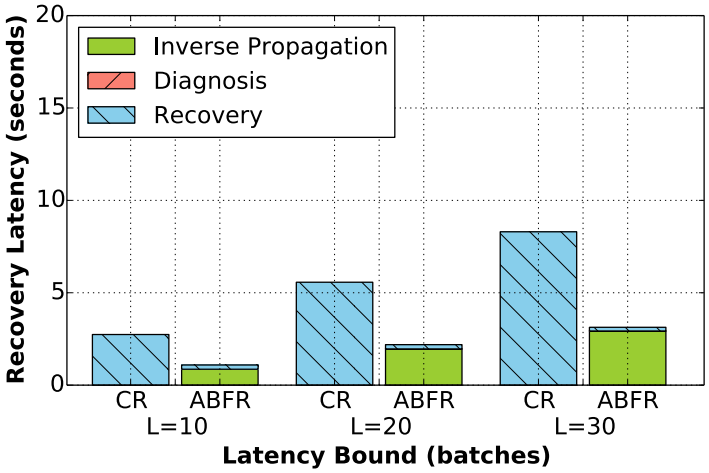


Figure 5.13: Strong scaling, MC Particle Transport Recovery Latency: ABFR vs. CR

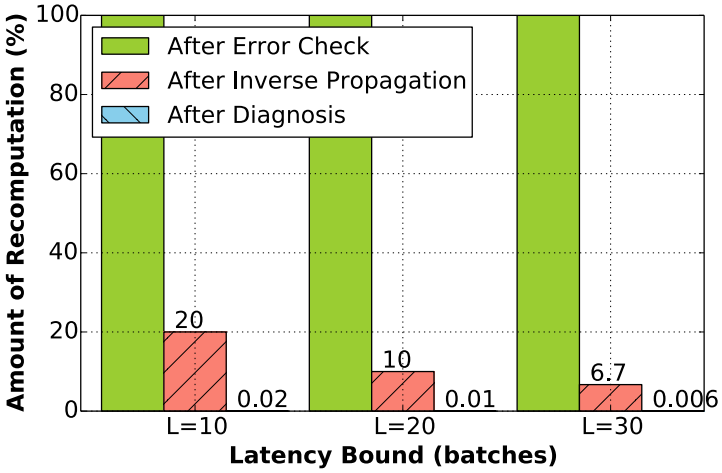


Figure 5.14: Strong scaling, amount of recomputation needed after each operator

after inverse propagation is 20%, 10% and 6.7%. After diagnosis, this number is further reduced to 0.02%, 0.01% and 0.006%. The results demonstrate that with strong scaling, ABFR still effectively focuses recovery and improves the recovery performance.

Weak Scaling. Each process simulates the same number of particles, i.e. 1000 particles. We scale up the number of processes, which means the job solves a larger problem (1024x1024 particles).

Figure 5.15 compares the recovery cost of ABFR and CR, and Figure 5.15 compares the recovery latency of ABFR and CR for error latency bounds from 10 to 30 batches. ABFR

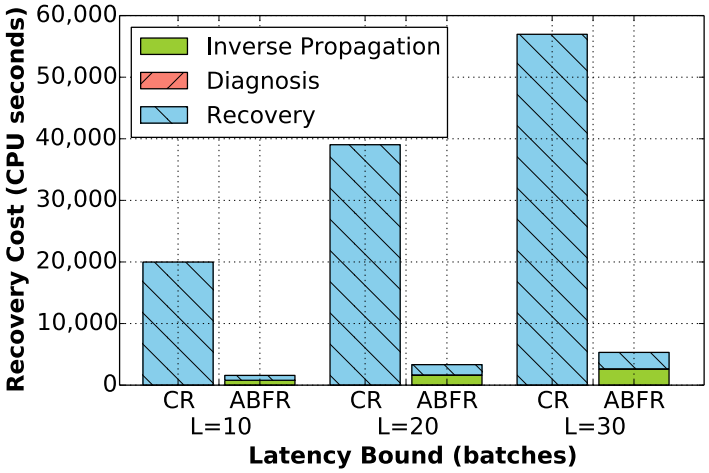


Figure 5.15: Weak scaling, MC Particle Transport Recovery Cost: ABFR vs. CR

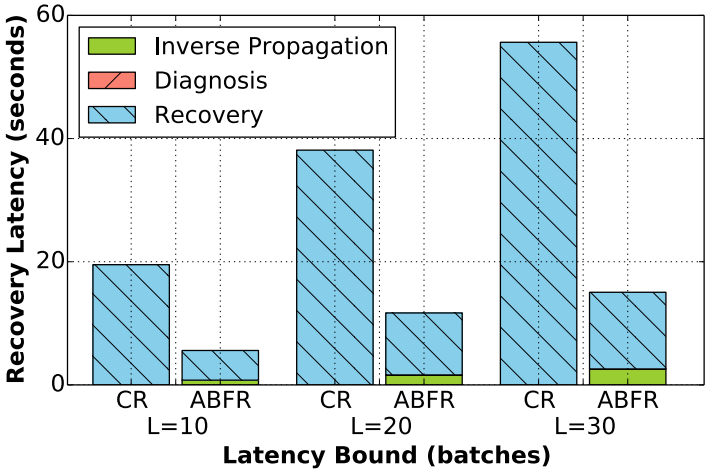


Figure 5.16: Weak scaling, MC Particle Transport Recovery Latency: ABFR vs. CR

outperforms CR by 23x in recovery cost and 3.5x in recovery latency. The reason that the benefit ratio is decreased compare to 128-proc run is because there are more PRC batches in 1024-proc run. With 128 proc, we have 2 PRC batches at latency 10, 4 PRC batches at latency 20, and 10 PRC batches at latency 30. However, with 1024 process, there are 8 PRC batches at latency 10, 18 PRC batches at latency 20, and 28 PRC batches at latency 30. In experiment configuration, the simulations have the same size of tallies for weak scaling, which means the tally scores are calculated for the same size of geometry space. When we scale the number of processes and particles, the simulation space becomes more dense. The

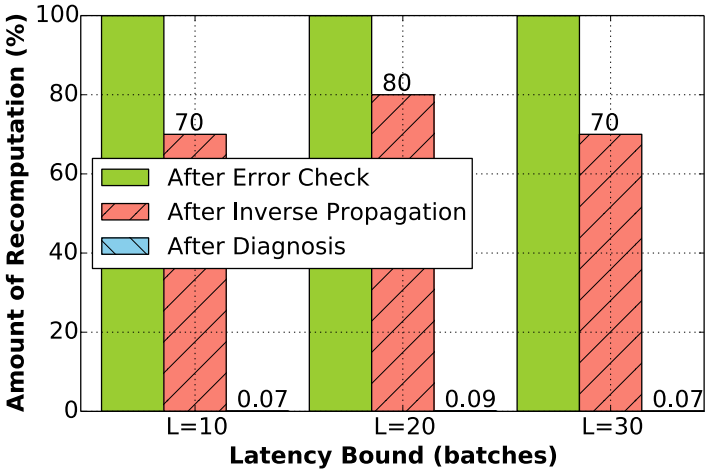


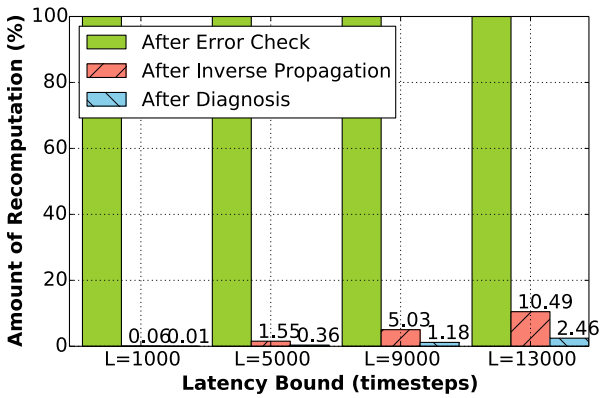
Figure 5.17: Weak scaling, amount of recomputation needed after each operator

number of events to be scored become more frequent, increasing the density of tally arrays and therefore the number of PRC batches. On the other hand, even though there are more PRC batches, the number of PRC processes that actually contributed to the erroneous tally is only 2 in all cases. In summary, the inverse propagation reduces the recovery scope to 8 - 28 PRC batches, only removing recomputation needed by 20% to 7%. But the diagnosis works efficiently, further reducing the recovery scope on to 2 processes out of 1024 processes. The resulting recomputation needed is less than 0.18%.

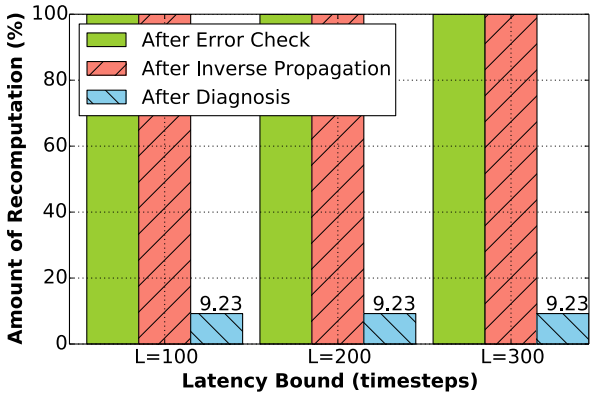
Our results demonstrate that ABFR effectively reduces recovery cost both in strong scaling and weak scaling by focusing recovery work on where needed. The scalability of ABFR suggests it is a promising direction of fault tolerance in high error rate extreme-scale systems.

5.5 Effective Recovery Focus

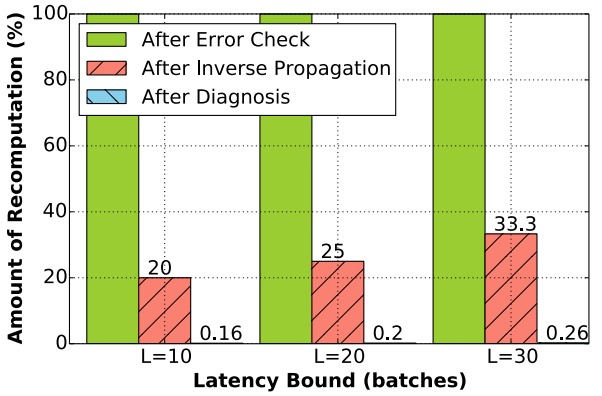
ABFR reduces recovery cost and recovery latency dramatically in all of the computational archetypes. This reduction is the direct result of inverse propagation and diagnosis that together enable focus of recovery effort. For all three computations, Figure 5.18 shows how the required recomputation shrinks with each operator. After error check, the needed



(a) Stencil



(b) N-Body



(c) MC Particle Transport

Figure 5.18: Achieved Recovery Focus: Operator impact on recovery work

recomputation is the full computation breadth times the error latency bound (i.e. 100%). CR recomputes all data for the latency bound, therefore it is 100% in comparison.

For stencil (Figure 5.18a) inverse propagation reduces the amount of recomputation re-

quired from 100% to 0.06% for 1,000 timesteps error latency bound and to 10.49% for 13,000 timesteps error latency bound. Diagnosis further cuts the needed recomputation to 0.01% and 2.46% respectively. In N-Body (see Figure 5.18b), inverse propagation has little reductive impact; diagnosis is the main contributor to focusing recovery, cutting the recovery effort to 9.23% of the original amount. The MC particle transport experiment localizes error scope from all the batches to a few batches just by inverse propagation. Correspondingly, the amount of recomputation is reduced to 20% to 33.3%. The pruning of PRC processes in diagnosis narrows the recovery recomputation to only 0.16% to 0.26%.

For all the studies, inverse propagation is a lightweight operator, refining recovery scope coarsely, but can still be effective in reducing its size. Inverse propagation can exploit predictable dataflow and communication, or as in the case of OpenMC, may require the addition of new data structures to record actual data flow. Diagnosis operators often include some analysis, and have also proven effective in pruning PRCs. Diagnosis further reduced recovery computation to 2.46% (stencil), 9.23% (N-Body), and 0.26% (Monte Carlo). Stencil shows that "recompute and compare" can be effective, but costly. Exploiting more application knowledge, such as opening criteria in N-Body and using a bit-map to track potential error propagation can increase effectiveness at low cost.

5.6 Discussion and Summary

To summarize, our results show that ABFR outperforms CR by a large margin in all three applications, reducing latent-error recovery-cost by 2.4x to 367x and recovery latency by 2.2x to 24x. Benefits are derived primarily from intelligence in inverse propagation and diagnosis operators that focus recovery on the actual corrupted data; in some cases we achieve >200-fold reduction in recovery effort.

ABFR operators require computation, but our experience shows that computational effort for inverse propagation is small for all three applications, and diagnosis is small for two of the three. For stencil, we believe diagnosis cost can be reduced further. Regardless, the

modest cost of these operators means that the benefits of focused recovery produce much lower recovery cost overall.

We only study recovery cost here, but ABFR in a production application setting would incur additional overheads: periodic versioning, tracking structures such as in OpenMC. Measurements indicate that these overheads are negligible. In OpenMC, the bit-vector tracking that supports inverse propagation and diagnosis increase runtime by less than 0.02%. Our prior work on GVR showed that versioning at the frequencies used in our studies can be achieved with overheads <1% [35]. We have also shown that using optimal intervals, the ABFR overhead is <1% for wide variety of error rates, while CR overhead increases quickly with error rates, increasing to >10% in the same range [50].

Our experiments are at significant scale, but scientific applications can be much larger. We expect ABFR's benefits to scale up with application size because ABFR focuses recovery effort only where needed. So, its cost depends mostly on error latency bound and latent-error rate, not application size. Better ABFR operators, employing expert knowledge, and ABFR parallelism, can increase benefits. For instance, in stencil, diagnosis latency is improved by parallelization and load balance. More sophisticated operators can further improve performance. Further, in MC particle transport, simply removing corrupted tally values (forward recovery) [34], and more samples may suffice to meet the convergence condition.

CHAPTER 6

AUTOMATING ABFR

In this chapter, we discuss an approach to automatically support ABFR in applications. ABFR defines the application knowledge needed for efficient latent error recovery [52]. Such knowledge is often shared across the same class of computations. Therefore there is opportunity to build ABFR operators into a domain specific library for flexible reuse, alleviating programmer effort. We experimented with a compiler-based approach to automatically add ABFR and GVR functions in stencil computations and demonstrated efficient and scalable recovery.

6.1 Enable ABFR in Stencil Computations Through ROSE-ABFR Translator

We present a compiler-based approach to automatically add ABFR functions and GVR versioning functions [35] to perform ABFR resilience for stencil computations in a simplified and portable way. [1].

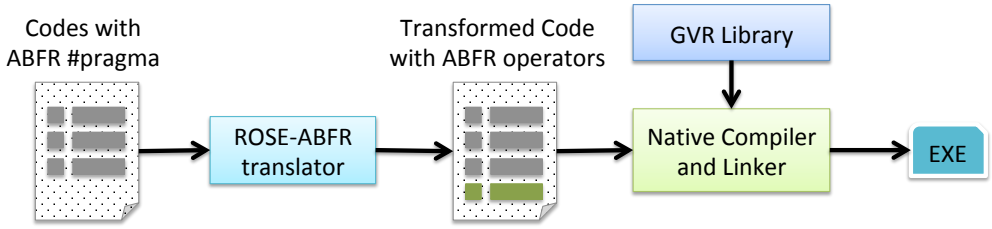


Figure 6.1: Diagram of Using ROSE-ABFR Translator to Automate ABFR in Applications.

We implement a translator using the ROSE source-to-source compiler infrastructure [78]. We define five pragmas for users to provide application information and specify where in the application to apply transformation. The translator parses source code, recognizing pragmas and adding ABFR/GVR functions correspondingly. A vendor compiler generates

1. Currently the translator only supports stencil codes writting in C language.

executables from the transformed code. Figure 6.1 illustrates how to use ROSE-ABFR translator to automate ABFR in application source codes. The library is released in [49], including source codes and an example of ABFR-augmented stencil computations.

We apply the ROSE-ABFR translator on a 2-D heat equation application and show that the translator automatically adds ABFR/GVR functions appropriately. We evaluate the recovery performance of the transformed stencil application on NERSC Edison supercomputer. The results show that the recovery cost is less than 1.2% for scales up to 1024 processes, demonstrating that ROSE-ABFR can achieve portable, scalable and efficient recovery performance.

6.2 Design of ROSE-ABFR Translator

ABFR defines a general framework for application-based recovery: four basic operators (error check, inverse propagation, diagnosis and recovery) are implemented by developers using application knowledge (refer to [50] for more details). The ROSE-ABFR translator automatically adds GVR versioning functions (for backup of data) and ABFR recovery functions (including diagnosis and recovery operators). Therefore developers only need to implement error check and inverse propagation. The interfaces of two operators are defined as follows.

```
1 // return the error index if an error is found
2 int *Error_Check(void * data);
3
4 // return indexes of a set of potential root causes.
5 int **invert_propagation(int *error_index, int step, int *nPRC,
int *ABFR_PRC_RANK);
```

Figure 6.2: Interfaces of Error Check and Inverse Propagation operators

The error check operator verifies the states of application data and returns the error index if an error is detected. Given the error information (`error_index`) and time (`step`), the inverse propagation operator exploits application data flow to identify potential root causes (`PRCs`) of the detected error. It sets the number of potential root causes `nPRC` and

Table 6.1: ABFR Pragmas

Pragma	Attributes	Semantics
`#pragma abfr_gvr_init argc argv`	argc: the number of command line arguments, argv: command line arguements	Initialize GVR and ABFR. Called forefront.
`#pragma abfr_gvr_finalize`	-	Finalize GVR and ABFR in the end.
`#pragma abfr_init_version data ndims count type gds`	data: data to protect, ndims: dimension of stencil, count: number of elements in each dimension, type: data type, gds: name for GVR handler.	Initialize GVR versions for data backup
`#pragma abfr_versioning data ld lo_index hi_index gds`	data: data to create version, ld: defines shape of local buffer, lo_index: starting element of array, hi_index: ending element of array, gds: name of GVR handler,	Create a version (snapshot) of data
`#pragma abfr_recover data gds error_index step latency interval error_tol invert_propagation compute`	data: data to recover, gds: name of GVR handler, error_index: error index in array, step: current time-step, latency: error latency bound, interval: versioning interval, error_tol: acceptable error tolerance, invert_propagation: function of inverse propagation, compute: function to compute stencil elements	Invoke ABFR recovery function

indicates if the calling process contains PRCs by setting `ABFR_PRC_RANK`.

We use GVR library to persist data. GVR's low-cost versioning enables flexible recovery for ABFR. Please refer to [1] for guides of installation and usage.

We define five pragmas which developers can use to specify where to add ABFR and GVR functions in applications and also provide information of stencils (e.g. data type, number of dimensions, size in each dimension, etc).

- `#pragma abfr_gvr_init` initializes ABFR and GVR. It should be placed at the beginning of main function, before calling other ABFR functions.
- `#pragma abfr_gvr_finalize` frees allocated data and finalizes. It should be placed in the end. No other ABFR/GVR functions should be called after this pragma.
- `#pragma abfr_init_version` allocates memory for GVR array to persist data. It requires information of stencils.
- `#pragma abfr_versioning` creates a snapshot of the data. Users can specify the versioning frequency by wrapping it in a `if` statement. E.g. `if(step % interval) == 0`.
- `#pragma abfr_recover` invokes ABFR function, which calls inverse propagation to identify potential root causes, diagnose and prune PRCs and recompute PRCs.

The definition of pragmas are given in Table 6.1. ROSE-ABFR translator parses pragmas and adds ABFR/GVR functions respectively.

6.3 Example of 2-D Stencil

We apply ABFR pragmas to a 2-D heat equation code and explain how the ROSE-ABFR translator works.

First, we add `abfr_gvr_init` in the beginning of the main function in Figure 6.3. Figure 6.4 shows the output of the translator, that a GVR initailize function is added.

```
74 #pragma abfr_gvr_init argc argv
75     MPI_Init(&argc,&argv);
76     MPI_Comm_size(MPI_COMM_WORLD,&numtasks);
77     MPI_Comm_rank(MPI_COMM_WORLD,&taskid);
```

Figure 6.3: pragma abfr_gvr_init

```
96 #pragma abfr_gvr_init argc argv
97   GDS_thread_support_t provd_support;
98   GDS_init(&argc,&argv,GDS_THREAD_MULTIPLE,&provd_support);
99   MPI_Init(&argc,&argv);
```

Figure 6.4: Output of ROSE-ABFR translator for pragma abfr_gvr_init

We add `abfr_gvr_finalize` in the end of main function as shown in Figure 6.5. Figure 6.6 shows the output. GVR finalize function is called after pragma.

```
246 #pragma abfr_gvr_finalize
247     MPI_Finalize();
```

Figure 6.5: pragma abfr_gvr_finalize

```
272 #pragma abfr_gvr_finalize
273   GDS_free(&gds_u);
274   GDS_finalize();
275   MPI_Finalize();
276 }
```

Figure 6.6: Output of ROSE-ABFR translator for pragma abfr_gvr_finalize

To allocate GVR array, we specify the data structure to protect, i.e. `u`, the number of dimension of stencil in `ndims` variable and the size of each dimension in `count` variable, the data type `double` and a user-define GVR handler `gds_u` as shown in Figure 6.7. Figure 6.8 illustrates the added code for allocating GVR array.

```
119     int ndims = 2;
120     int count[2] = {NX, NY};
121 #pragma abfr_init_version u ndims count double gds_u
```

Figure 6.7: pragma abfr_init_version

To create a version (snapshot) of the array periodically, we wrap the pragma in a `if` statement as shown in Figure 6.9. The output of translator is shown in Figure 6.10. It adds GVR put and versioning function calls.

```
153 #pragma abfr_init_version u ndims count double gds_u
154   int abfr_i;
155   GDS_size_t *min_chunks;
156   min_chunks = ((GDS_size_t *)(calloc(sizeof(GDS_size_t ),ndims)));
157   for (abfr_i = 0; abfr_i < ndims; ++abfr_i)
158     min_chunks[abfr_i] = 0;
159   GDS_gds_t gds_u;
160   MPI_Info abfr_info;
161   MPI_Info_create(&abfr_info);
162   GDS_datatype_t abfr_type = GDS_DATA_DBL;
163   GDS_alloc(ndims,count,min_chunks,abfr_type,GDS_PRIORITY_HIGH,GDS_COMM_WORL
    D,abfr_info,gds_u);
```

Figure 6.8: Output of ROSE-ABFR translator for pragma abfr_init_version

```
217         size_t ld[1] = {NY};
218         size_t lo_index[2] = {(NX/numtasks)*taskid,0};
219         size_t hi_index[2] = {(NX/numtasks)*(taskid+1)-1,NY-1};
220         if(it % interval == 0) {
221 #pragma abfr_versioning u ld lo_index hi_index gds_u
222             if(taskid == MASTER) printf("versioning at step %d\n", it);
223         }
```

Figure 6.9: pragma abfr_versioning

```
253 #pragma abfr_versioning u ld lo_index hi_index gds_u
254       GDS_put(u,ld,lo_index,hi_index,gds_u);
255       GDS_version_inc(gds_u,1,'\0',0);
256       if (taskid == 0)
257         printf("versioning at step %d\n",it);
258     }
```

Figure 6.10: Output of ROSE-ABFR translator for pragma abfr_versioning

Figure 6.11 shows the usage of `abfr_recover` pragma. Information such as data, GVR handler, error index, timestep, error latency bound, versioning interval, error tolerance threshold, inverse propagation function and compute function are provided in the pragma. A `abfr_recovery` function is added in the source file and the function call is appended after the pragma, as illustrated in Figure 6.12.

```
210           int latency;
211           int interval;
212           double error_tol = 0.001;
213 #pragma abfr_recover u gds_u error_index it latency interval error_tol invert_propagation compute
```

Figure 6.11: pragma abfr_recover

Please refer to the source codes in example directory for detailed usage of pragmas.

```
238
239 #pragma abfr_recover u gds_u error_index it latency interval error_tol invert_propagation compute
240       abfr_recovery(u,gds_u,ndims,count,ld,lo_index,hi_index,it,error_index,latency,interval,error_to
    l);
```

Figure 6.12: Output of ROSE-ABFR translator for pragma abfr_recover

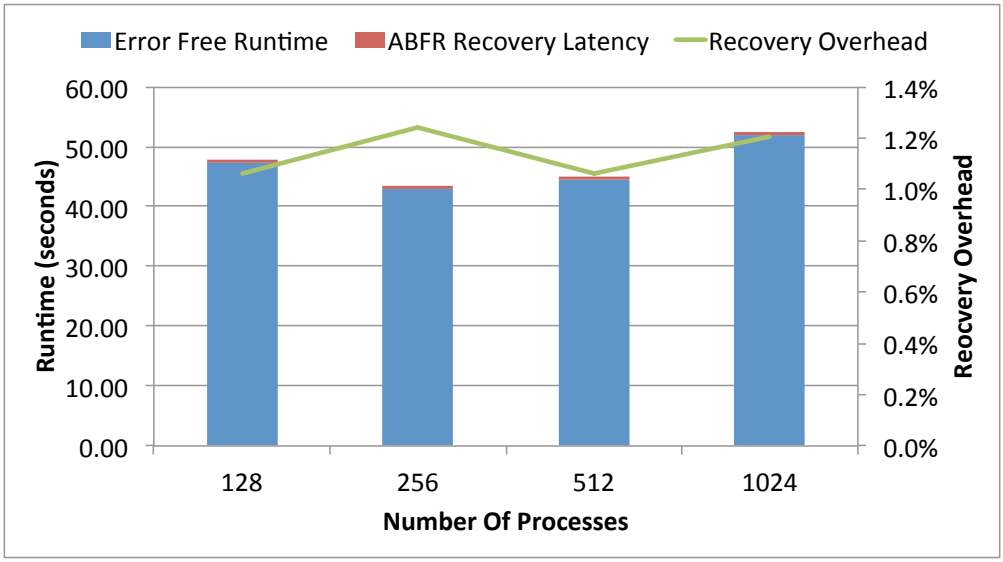


Figure 6.13: Recovery overhead for varied scales

6.4 Experiments: Performance Evaluation

We evaluate the performance of the ROSE-ABFR transformed heat equation codes for varied scales (128, 256, 512, 1024 processes) and error latencies (500 to 4000 timesteps).

First we run same amount of work per process (weak-scaling) and vary the number of process from 128 to 1024. We inject an error in the middle of run and measure the recovery latency (runtime). The error latency bound (intervals between two error checks) is 1000 timesteps. The experiment configurations are shown in Table 6.2.

Table 6.2: Experiment Configuration for Varied Scales

Number of processes	128, 256, 512, 1024
Stencil size per process	8192 * 8192
Total timesteps	10,000
Error latency	1000

Figure 6.13 shows the recovery cost for varied scales. The error free runtime is around 47

Table 6.3: Experiment Configuration for Varied Error Latencies

Number of processes	1024
Stencil size per process	8192 * 8192
Error latency	500,1000,1500,2000,2500,3000,3500,4000

seconds for 10,000 timesteps, about 0.0047 seconds per timestep. While the recovery cost for error latency of 1000 timesteps is less than 1.2% for all scales, including restoring data and recovery. The results validate that the transformed application can achieve efficient recovery performance with ABFR support.

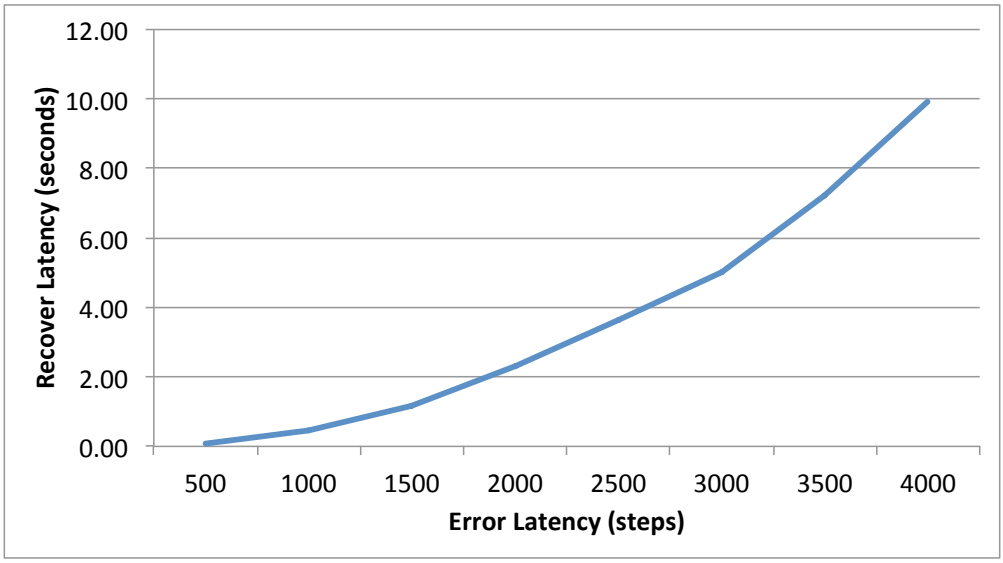


Figure 6.14: Recovery overhead vs. Error Latency

Second, we explore the recovery performance for varied error latencies. Experiment configuration is listed in Table 6.3.

Figure 6.14 illustrates the recovery cost for varied error latencies. The results show that the recovery cost starts slow and grows polynomially with error latencies. ABFR focuses recovery only on corrupted data. As the error latency scales up, more and more data are corrupted (error propagates to four neighbors in one step in 2-D stencil), therefore the recovery cost increases polynomially with error latencies. The results demonstrate that the transformed application achieves scalable performance for varied error latencies.

The ROSE-ABFR translator reduces the number of codes required to apply ABFR resilience and allows users to focus on the application, thus providing higher productivity and portability with an easy-to-use interfaces.

CHAPTER 7

CONCLUSIONS AND FUTURE DIRECTIONS

7.1 Summary

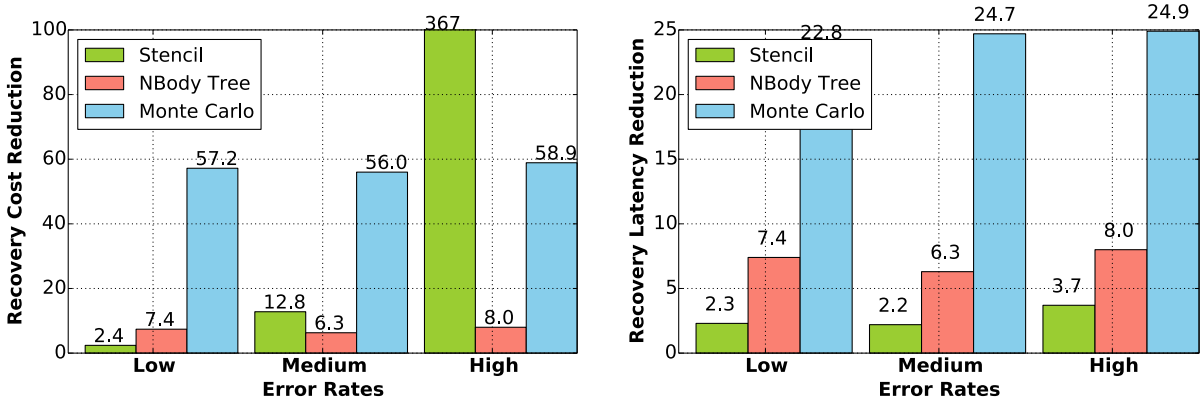
High error rates predicted in future exascale systems is a critical challenge to applications. Latent errors threaten the validity of computational results, limiting the scale of science. In the dissertation, we presented **A**pplication-**B**ased **F**ocused **R**ecovery (ABFR), that exploits application knowledge to focus recovery effort. ABFR enables application designers to express flexible, application-based resilience for latent errors. The key to ABFR is encapsulating the needed application knowledge in four operators, separating the concerns of application semantics and underlying runtime parallelization and overlap. The dissertation described the ABFR operator definition and interfaces, providing a general framework on how to exploit application knowledge for latent error resilience. We presented an ABFR runtime that intelligently exploits parallelism and GVR global view to achieve load balancing and implement the complex recovery procedure. ABFR focuses recovery on where needed, making it a promising direction for addressing latent errors in high-error rates large-scale systems.

Table 7.1: Summary of Code Changes in Applying ABFR

Applications	Lines of Code Added	Application Lines of Code	%
Chombo	894	500K	0.2%
Gadget2	870	20K	4.3%
OpenMC	420	32K	1.3%

Application to three diverse archetypes demonstrates the breadth of the ABFR's approach, and illustrates both what is involved in real ABFR operators, and the numerous options for flexible ABFR for latent errors for each archetype. The three computations are widely used in science areas but vary significantly in algorithms, communication pattern,

and data structure. The experience of applying of ABFR to them suggest that (1) limited application knowledge is required; (2) there are flexible choices in the ABFR operator design, from simply following original data flow to adding data structures. Application designers can easily experiment with a range of strategies. Table 7.1 documents the lines of code added in applying ABFR to three application codes (Chombo heat equation, Gadget2, OpenMC). Our studies show that the code changes are less than 1.3% for three production applications.



(a) Recovery Cost Reduction

(b) Recovery Latency Reduction

Figure 7.1: ABFR significantly reduces recovery cost for all three computation archetypes

Experiments with application codes (Chombo heat equation, Gadget2, OpenMC) demonstrate the scalability and efficiency of ABFR runtime and overall ABFR recovery. Figure 7.1 summarizes the performance of ABFR in reducing recovery cost and latency for three computation archetypes. In general, ABFR achieves the greatest performance improvement for the short error latency bound scenarios (see Chapter 5), but significant benefits for all error latency bounds. The analytical model built in Chapter 4 shows that increasing error rates would shrink the optimal error check interval (i.e. error latency bound), as illustrated in Figure 4.4. We map error latency bounds to error rates in Figure 7.1. ABFR reduces recovery cost by 367x for stencil, 8x for N-Body tree and 58.9x for Monte Carlo, outperforming CR in high error rates environment. Note that these results may be improved by more sophisticated application ABFR operators. The results demonstrate that ABFR achieves scalable recovery performance on large-scale systems.

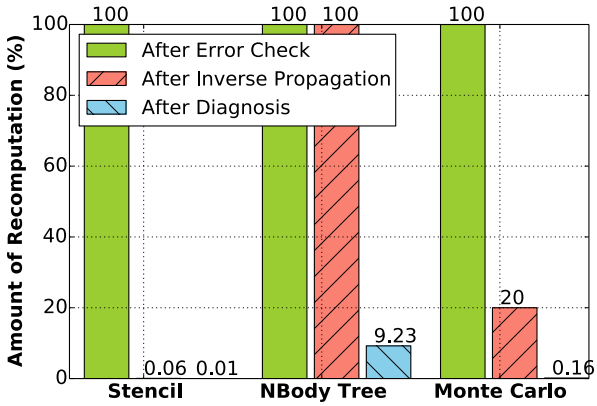


Figure 7.2: ABFR effectively focuses recovery in high error rates: operator impact on recovery work

The benefits of ABFR come directly from inverse propagation and diagnosis operators. Figure 7.2 summarizes the effectiveness of ABFR operators on reducing recovery scope for high error rates (i.e. short error latency bound). In all three cases, ABFR successfully focus recovery effort where it is needed.

Overall, the dissertation demonstrated that ABFR is an efficient and scalable application-based resilience approach for large-scale systems with high error rates.

7.2 Landscape of ABFR

ABFR arises as a new approach for latent error recovery. There are two questions to consider: (1) How general is ABFR? and (2) Does ABFR represents a new way forward?

1. How general is ABFR? To demonstrate the generality of ABFR, we studied three diverse computation archetypes which are widely used in materials science, fusion energy and chemistry. They represent a challenging set, varying in communication pattern, data structure and application recovery strategy. Our experience shows that only limited application knowledge is required in ABFR operator design. Some of these knowledge is already managed by developers as part of the applications. Essentially, no knowledge of the underlying system is required. While we studied three representative archetypes, stencil, N-Body tree,

Monte Carlo, we believe ABFR is applicable to a considerable number of computations. We identify several application properties and error properties that ABFR is most applicable to. But these properties are not necessities.

- **Localizable Error Propagation.** Applications that have regular data dependencies, such as stencils and adaptive mesh refinement (AMR) can easily adopt ABFR to bound error effect and focus recovery. Some applications have dependency tables or graphs that can be exploited by ABFR. Such examples include broad graph processing algorithms and task-parallel applications. Some applications have properties that limit the spread of errors. For instance, N-Body tree codes have numerical cut-off that confine erroneous regions to some subtrees. Monte Carlo applications do not propagate errors across sampled batches. These application all localize error effect. In addition, ABFR enables parallel diagnosis and recovery, exploiting global view and multi-versions provided by GVR. The runtime of recovery is further improved.

- **Forward Correctable.** For some applications, errors are not so easily or purely localized but they are forward correctable. This can be either a precise correction or an approximate correction. For instance, applications with strong convergence properties can forward computation and produce correct results [35]. ABFR supports such forward correction in recovery operator.

- **Hard to Detect Latent Errors.** These errors are difficult to capture and usually detected a long time after their occurrence, also known as silent data corruption. We have discussed the three challenges: (1) when the error occurred? (2) what data was corrupted? and (3) how to recovery efficiently? ABFR solves each challenge with a well defined framework. Applications are able to bound the error latency and focus recovery effort where needed. The cost of ABFR is determined by the error latency and application semantics, rather than depending on the application scale or problem size. Therefore ABFR is efficient and scalable for tolerating such latent errors.

The properties identified above are a subset. But some applications may benefit less from ABFR. For example, some applications have instant global error propagation, where single function call spreads the error across all the data, leaving no opportunity for focusing recovery. In such cases, a simplified version of ABFR (without inverse propagation and diagnosis) can always be applied to achieve resilience. Essentially, ABFR subsumes CR and MCR. We believe ABFR is general and applicable to large sets of applications.

2. Does ABFR represent a new way forward? In Chapter 2, we presented two directions to solve the resilience problem: CR and ABFT. We have compared the performance of ABFR with CR in Chapter 5. In all studies, CR's cost grows linearly with the error latency and application scale, producing high overheads. ABFR solves these CR scalability problems by exploiting application knowledge to focus recovery on where needed. The cost of ABFR is determined by the error latency and application semantics, rather than depending on the application scale or problem size. Therefore ABFR is much more efficient and scalable, making it suitable for future large-scale systems with high error rates.

ABFT is promising and demonstrated to incur low recovery overhead for some algorithms. However, there are no general principles or models for applying ABFT to applications. In contrast, ABFR provides a general framework on how to design efficient resilience using application knowledge. Most ABFT techniques can be converted and adopted by ABFR with no additional cost. The algorithms designed for error detection and correction are intrinsically part of error check and recovery operators. ABFR provides application designers the flexibility to express application-based resilience using a range of application knowledge. The benefit of conversion is the portability and the ability to easily experiment with and optimize across varied strategies and designs. ABFR is more general and achieves better latent error recovery performance compared to the state of art ABFT techniques.

Through the dissertation, we demonstrated that ABFR expands the scope of resilience as illustrated in Figure 1.6. Three computation archetype studies show that applications are

able to tolerate long error latencies with ABFR's efficient and scalable resilience support. We believe ABFR represents a new way forward.

7.3 Future Directions

Breadth of ABFR. While we have studied three very different application archetypes, to increase the breadth of experience with ABFR, further application studies with ABFR would be valuable. Further application experiments both in the design of a variety of ABFR operators and large-scale empirical studies are needed to substantiate and document the breadth of applicability of the ABFR approach.

Application of ABFR to Non-SPMD, Non-bulk synchronous Applications. We have demonstrated ABFR on three computation archetypes. It is interesting to explore the potential of ABFR on non-SPMD, non-bulk synchronous applications, such as task-based applications, databases. These applications differ in data structures, algorithms, synchrony and computation flows. How to apply focused recovery through ABFR operators is an interesting question.

Optimal Error Checking Interval. Error checking and versioning each has a cost and thus impact the performance of applications. Determining the best frequency of error checking and versioning is a critical problem, which is in general difficult in face of the more complex recovery costs. Following Young and Daly's [98, 38] effort to optimal checkpoint interval, we built analytical models with key factors taken into account and derive the optimal error checking and versioning intervals for stencil computations. Creation of analytic models to select intervals for other computations are desired. These models are specifically designed for each ABFR design and recovery behavior. Can a unified model to select optimal error checking and versioning intervals for latent errors be created?

Multiple Latent Errors. For simplicity we only experimented with single errors. This assumption is common and underlies much of Checkpoint-Restart practice. But nothing in the ABFR approach limits it to single errors. There are several potential avenues for extension.

First, multiple errors within a detection interval could trigger multiple ABFR responses. The inverse propagation, diagnosis and recovery operators work on each error respectively. That is, each error triggers an ABFR recovery. Alternatively, diagnosis and recovery could be extended to deal with multiple errors concurrently. The inverse propagation operator returns the potential root causes of all detected errors. The input to diagnosis and recovery is therefore a set of complete PRCs, so they can perform simultaneously for all error sources. Separate or concurrent operators can be controlled by designers in implementation. These are promising directions for future work.

Reducing Programmer Effort. While our work shows that ABFR can be applied with reasonable effort and with portability, but reducing programmer effort is always worth exploring. We have experimented with a compiler-based approach to automatically add ABFR and GVR functions in stencil computations and demonstrated efficient and scalable recovery. There are other possibilities across a range of applications. For instance, building ABFR into domain-specific language (DSL) is a potential direction. DSL is specialized to a particular application domain. Application designers can easily apply ABFR when coding with DSLs.

BIBLIOGRAPHY

[1] Global View Resilience. http://people.cs.uchicago.edu/ aachien/lssg/research/gvr/.

[2] Aurora Supercomputer Will Be America's First Exascale System. https://www.top500.org/news/retooled-aurora-supercomputer-will-be-americas-first-exascale-system/, 2017.

[3] NERSC CORI Supercomputer. https://www.nersc.gov/users/computational-systems/cori/, 2017.

[4] NERSC Edison. https://www.nersc.gov/users/computational-systems/edison/, 2017.

[5] RCC Midway High-Performance Computing. https://rcc.uchicago.edu/resources/high-performance-computing, 2017.

[6] Summit, Oak Ridge National Laboratory's next High Performance Supercomputer. https://www.olcf.ornl.gov/summit/, 2017.

[7] TIANHE-2. https://www.top500.org/featured/systems/tianhe-2/, 2017.

[8] Blue Waters Supercomputer. http://www.ncsa.illinois.edu/enabling/bluewaters, 2018.

[9] Jülich Supercomputing Centre (JSC). http://www.fz-juelich.de/ias/jsc/EN/Expertise/Supercomputers/JUQUEEN/JUQUEEN_node.html, 2018.

[10] NERSC: Benchmark & Workload. http://www.nersc.gov/research-and-development/benchmarking-and-workload-characterization/, 2018.

[11] Sierra High Performance Computing System. https://computation.llnl.gov/computers/sierra, 2018.

[12] Sunway TaihuLight Supercomputer. https://www.top500.org/system/178764, 2018.

[13] Sverre J Aarseth and Sverre Johannes Aarseth. *Gravitational N-body simulations: tools and algorithms*. Cambridge University Press, 2003.

[14] GJ Ackland, G Tichy, V Vitek, and MW Finnis. Simple n-body potentials for the noble metals and nickel. *Philosophical Magazine A*, 56(6):735–756, 1987.

[15] Yves Robertand Andrew Chien Aiman Fang, Aurelien Cavelan. Resilience for stencil computations with latent errors (extended report). Research report RR-9042, INRIA, 2017.

[16] Andrew W Appel. An efficient program for many-body simulation. *SIAM Journal on Scientific and Statistical Computing*, 6(1):85–103, 1985.

[17] G. Aupy, A. Benoit, T. Hérault, Y. Robert, F. Vivien, and D. Zaidouni. On the combination of silent error detection and checkpointing. In *2013 IEEE 19th Pacific Rim International Symposium on Dependable Computing*, pages 11–20, Dec 2013.

[18] Algirdas Avizienis, J-C Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE transactions on dependable and secure computing*, 1(1):11–33, 2004.

[19] Josh Barnes and Piet Hut. A hierarchical o (n log n) force-calculation algorithm. *nature*, 324(6096):446–449, 1986.

[20] Leonardo Bautista-Gomez, Seiji Tsuboi, Dimitri Komatitsch, Franck Cappello, Naoya Maruyama, and Satoshi Matsuoka. Fti: high performance fault tolerance interface for hybrid systems. In *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for*, pages 1–12. IEEE, 2011.

[21] RC Bending. Direction-dependent exponential biassing. Technical report, 1974.

[22] Austin R Benson, Sven Schmit, and Robert Schreiber. Silent error detection in numerical time-stepping schemes. *The International Journal of High Performance Computing Applications*, 29(4):403–421, 2015.

[23] Eduardo Berrocal, Leonardo Bautista-Gomez, Sheng Di, Zhiling Lan, and Franck Cappello. Lightweight silent data corruption detection based on runtime data analysis for hpc applications. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, pages 275–278. ACM, 2015.

[24] George Bosilca, Rémi Delmas, Jack Dongarra, and Julien Langou. Algorithm-based fault tolerance applied to high performance computing. *Journal of Parallel and Distributed Computing*, 69(4):410–416, 2009.

[25] Marin Bougeret, Henri Casanova, Yves Robert, Frédéric Vivien, and Dounia Zaidouni. Using group replication for resilience on exascale systems. *The International Journal of High Performance Computing Applications*, 28(2):210–224, 2014.

[26] Judith F Briesmeister et al. Mcnptm-a general monte carlo n-particle transport code. *LA-13709-M, Los Alamos National Laboratory*, page 2, 2000.

[27] Franck Cappello. Fault tolerance in petascale/exascale systems: Current knowledge, challenges and research opportunities. *IJHPCA*, 2009.

[28] Franck Cappello, Al Geist, Bill Gropp, Laxmikant Kale, Bill Kramer, and Marc Snir. Toward exascale resilience. 2009.

[29] Franck Cappello, Al Geist, William Gropp, Sanjay Kale, Bill Kramer, and Marc Snir. Toward exascale resilience: 2014 update. *Supercomputing frontiers and innovations*, 1(1):5–28, 2014.

[30] Franck Cappello, Al Geist, William Gropp, Sanjay Kale, Bill Kramer, and Marc Snir. Toward exascale resilience: 2014 update. *Supercomputing frontiers and innovations*, 1(1):5–28, 2014.

[31] Aurélien Cavelan, Aiman Fang, Andrew A Chien, and Yves Robert. Resilient n-body tree computations with algorithm-based focused recovery: Model and performance analysis. In *International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems*, pages 158–178. Springer, 2017.

[32] Chin-Long Chen and MY Hsiao. Error-correcting codes for semiconductor memory applications: A state-of-the-art review. *IBM Journal of Research and Development*, 28(2):124–134, 1984.

[33] Zizhong Chen. Online-abft: An online algorithm based fault tolerance scheme for soft error detection in iterative methods. In *PPoPP '13*, 2013.

[34] A Chien, P Balaji, P Beckman, N Dun, A Fang, H Fujita, K Iskra, Z Rubenstein, Z Zheng, R Schreiber, et al. Versioned distributed arrays for resilience in scientific applications: Global view resilience. *ICCS*, 2015.

[35] A Chien, Pavan Balaji, Nan Dun, Aiman Fang, Hajime Fujita, Kamil Iskra, Zachary Rubenstein, Ziming Zheng, Jeff Hammond, Ignacio Laguna, et al. Exploring versioned distributed arrays for resilience in scientific applications: global view resilience. *IJHPCA*, 2017.

[36] Vinay K Chippa, Srimat T Chakradhar, Kaushik Roy, and Anand Raghunathan. Analysis and characterization of inherent application resilience for approximate computing. In *Proceedings of the 50th Annual Design Automation Conference*, page 113. ACM, 2013.

[37] P Colella, DT Graves, TJ Ligocki, DF Martin, D Modiano, DB Serafini, and B Van Straalen. Chombo software package for AMR applications design document. Technical report, LBNL, 2009.

[38] J. T. Daly. A higher order estimate of the optimum checkpoint interval for restart dumps. *Future Gener. Comput. Syst.*, 2006.

[39] Kaushik Datta, Shoaib Kamil, Samuel Williams, Leonid Oliker, John Shalf, and Katherine Yelick. Optimization and performance modeling of stencil computations on modern microprocessors. *SIAM review*, 51(1):129–159, 2009.

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

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

[42] Peng Du, Aurelien Bouteiller, George Bosilca, Thomas Herault, and Jack Dongarra. Algorithm-based fault tolerance for dense matrix factorizations. volume 47, pages 225–234. ACM, 2012.

[43] Anshu Dubey, Hajime Fujita, Daniel T Graves, Andrew Chien, and Devesh Tiwari. Granularity and the cost of error recovery in resilient amr scientific applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 42. IEEE Press, 2016.

[44] Anshu Dubey, Hajime Fujita, Zachary Rubenstein, Brian Van Straalen, and Andrew A Chien. A case study of application structure aware resilience through differentiated state saving and recovery. In *European Conference on Parallel Processing*, pages 619–630. Springer, 2015.

[45] Nan Dun, Hajime Fujita, John R Tramm, Andrew A Chien, and Andrew R Siegel. Data decomposition in monte carlo neutron transport simulations using global view arrays. *The International Journal of High Performance Computing Applications*, 29(3):348–365, 2015.

[46] Nan Dun, Dirk Pleiter, Aiman Fang, Nicolas Vandenbergen, and Andrew A Chien. Multi-versioning performance opportunities in bgas system for resilience. In *International Conference on High Performance Computing*, pages 486–504. Springer, 2016.

[47] JW Eastwood and RW Hockney. Computer simulation using particles. *New York: Mc GrawHill*, 1981.

[48] James F Epperson. *An introduction to numerical methods and analysis*. John Wiley & Sons, 2013.

[49] Aiman Fang. ABFR Rose Project. https://bitbucket.org/aimanf/abfr_gvr_rose, 2017.

[50] Aiman Fang, Aurélien Cavelan, Yves Robert, and Andrew A Chien. Resilience for stencil computations with latent errors. In *Parallel Processing (ICPP), 2017 46th International Conference on*, pages 581–590. IEEE, 2017.

[51] Aiman Fang and Andrew Chien. Applying GVR to Molecular Dynamics: Enabling Resilience for Scientific Computations. Technical report, University of Chicago, 2014.

[52] Aiman Fang and Andrew A Chien. Abfr: convenient management of latent error resilience using application knowledge. In *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*, pages 27–39. ACM, 2018.

[53] Bo Fang, Panruo Wu, Qiang Guan, Nathan DeBardeleben, Laura Monroe, Sean Blanchard, Zhizong Chen, Karthik Pattabiraman, and Matei Ripeanu. Sdc is in the eye of the beholder: A survey and preliminary study. In *Dependable Systems and Networks Workshop, 2016 46th Annual IEEE/IFIP International Conference on*, pages 72–76. IEEE, 2016.

[54] Kurt Ferreira, Jon Stearley, James H Laros, Ron Oldfield, Kevin Pedretti, Ron Brightwell, Rolf Riesen, Patrick G Bridges, and Dorian Arnold. Evaluating the viability of process replication reliability for exascale systems. In *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for*, pages 1–12. IEEE, 2011.

[55] David Fiala, Frank Mueller, Christian Engelmann, Rolf Riesen, Kurt Ferreira, and Ron 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*, page 78. IEEE Computer Society Press, 2012.

[56] Marc Gamell, Keita Teranishi, Michael A Heroux, Jackson Mayo, Hemanth Kolla, Jacqueline Chen, and Manish Parashar. Local recovery and failure masking for stencil-based applications at extreme scales. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 70. ACM, 2015.

[57] Erol Gelenbe. A model of roll-back recovery with multiple checkpoints. In *Proc. 2nd Int. Conf. on Software Engineering*, 1976.

[58] Leslie Greengard and Vladimir Rokhlin. A fast algorithm for particle simulations. *Journal of computational physics*, 73(2):325–348, 1987.

[59] Richard W Hamming. Error detecting and error correcting codes. *Bell Labs Technical Journal*, 1950.

[60] Paul H Hargrove and Jason C Duell. Berkeley lab checkpoint/restart (blcr) for linux clusters. In *Journal of Physics: Conference Series*, volume 46, page 494. IOP Publishing, 2006.

[61] Wayne Hayes. *Efficient shadowing of high dimensional chaotic systems with the large astrophysical N-body problem as an example.* University of Toronto, 1995.

[62] J Eduard Hoogenboom, William R Martin, and Bojan Petrovic. The monte carlo performance benchmark test-aims, specifications and first results. In *International Conference on Mathematics and Computational Methods Applied to*, volume 2, page 15, 2011.

[63] Kuang-Hua Huang and J.A. Abraham. Algorithm-based fault tolerance for matrix operations. *IEEE Trans. Computers*, 1984.

[64] Junsheng Long, W Kent Fuchs, and Jacob A Abraham. Forward recovery using checkpointing in parallel systems. In *ICPP (1)*, pages 272–275, 1990.

[65] Charng-Da Lu. *Scalable Diskless Checkpointing for Large Parallel Systems.* PhD thesis, Champaign, IL, USA, 2005. AAI3199074.

[66] Guoming Lu, Ziming Zheng, and Andrew A Chien. When is multi-version checkpointing needed? In *Proceedings of the 3rd Workshop on Fault-tolerance for HPC at extreme scale*, pages 49–56. ACM, 2013.

[67] Robert E Lyons and Wouter Vanderkulk. The use of triple-modular redundancy to improve computer reliability. *IBM Journal of Research and Development*, 6(2):200–209, 1962.

[68] Kaisheng Ma, Xueqing Li, Jinyang Li, Yongpan Liu, Yuan Xie, Jack Sampson, Mahmut Taylan Kandemir, and Vijaykrishnan Narayanan. Incidental computing on iot nonvolatile processors. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 204–218. ACM, 2017.

[69] William R Martin. Challenges and prospects for whole-core monte carlo analysis. *Nuclear Engineering and Technology*, 44(2):151–160, 2012.

[70] Stephen LW McMillan and Sverre J Aarseth. An o (n log n) integration scheme for collisional stellar systems. *The Astrophysical Journal*, 414:200–212, 1993.

[71] Subhasish Mitra, Norbert Seifert, Ming Zhang, Quan Shi, and Kee Sup Kim. Robust system design with built-in soft-error resilience. *Computer*, 38(2):43–52, 2005.

[72] Adam Moody, Greg Bronevetsky, Kathryn Mohror, and Bronis R de Supinski. Design, modeling, and evaluation of a scalable multi-level checkpointing system. In *Proceedings of the 2010 ACM/IEEE international conference for high performance computing, networking, storage and analysis*, pages 1–11. IEEE Computer Society, 2010.

[73] Anthony Nguyen, Nadathur Satish, Jatin Chhugani, Changkyu Kim, and Pradeep Dubey. 3.5-d blocking optimization for stencil computations on modern cpus and gpus. In *High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for*, pages 1–13. IEEE, 2010.

[74] Jarek Nieplocha, Bruce Palmer, Vinod Tipparaju, Manojkumar Krishnan, Harold Trease, and Edoardo Apra. Advances, applications and performance of the global arrays shared memory programming toolkit. *The International Journal of High Performance Computing Applications*, 20(2):203–231, 2006.

[75] Akira Nukada, Hiroyuki Takizawa, and Satoshi Matsuoka. Nvcr: A transparent checkpoint-restart library for nvidia cuda. In *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pages 104–113. IEEE, 2011.

[76] Robert W Numrich and John Reid. Co-array fortran for parallel programming. In *ACM Sigplan Fortran Forum*, volume 17, pages 1–31. ACM, 1998.

[77] William Wesley Peterson and Daniel T Brown. Cyclic codes for error detection. *Proceedings of the IRE*, 1961.

[78] Dan Quinlan and Chunhua Liao. The rose source-to-source compiler infrastructure. In *Cetus users and compiler infrastructure workshop, in conjunction with PACT*, volume 2011, page 1, 2011.

[79] Arash Rezaei, Giuseppe Coviello, Cheng-Hong Li, Srimat Chakradhar, and Frank Mueller. Snapify: capturing snapshots of offload applications on xeon phi manycore processors. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*, pages 1–12. ACM, 2014.

[80] Paul K Romano and Benoit Forget. The openmc monte carlo particle transport code. *Annals of Nuclear Energy*, 51:274–281, 2013.

[81] Paul Kollath Romano. *Parallel algorithms for Monte Carlo particle transport simulation on exascale computing architectures*. PhD thesis, MIT, 2013.

[82] Zachary Rubenstein, Hajime Fujita, Ziming Zheng, and Andrew Chien. Error checking and snapshot-based recovery in a preconditioned conjugate gradient solver. *Technical Report TR-2013-11, Department of Computer Science, University of Chicago*, 2013.

[83] Bianca Schroeder and Garth A Gibson. Understanding failures in petascale computers. In *Journal of Physics: Conference Series*, volume 78, page 012022. IOP Publishing, 2007.

[84] Veeral Shah and Milind Borate. Method and apparatus for performing transparent in-memory checkpointing, February 28 2012. US Patent 8,127,174.

[85] Manu Shantharam et al. Fault tolerant preconditioned conjugate gradient for sparse linear system solution. ICS, 2012.

[86] Manu Shantharam, Sowmyalatha Srinivasmurthy, and Padma Raghavan. Characterizing the impact of soft errors on iterative methods in scientific computing. In *Proceedings of the international conference on Supercomputing*, pages 152–161. ACM, 2011.

[87] Vishal Chandra Sharma, Ganesh Gopalakrishnan, and Greg Bronevetsky. Detecting soft errors in stencil based computations. *Geophysics*, 48(11):1514–1524, 1983.

[88] Guochun Shi, Jeremy Enos, Michael Showerman, and Volodymyr Kindratenko. On testing gpu memory for hard and soft errors. In *Proc. Symposium on Application Accelerators in High-Performance Computing*, volume 107, 2009.

[89] Marc Snir, Robert W Wisniewski, Jacob A Abraham, Sarita V Adve, Saurabh Bagchi, Pavan Balaji, Jim Belak, Pradip Bose, Franck Cappello, Bill Carlson, et al. Addressing failures in exascale computing. *The International Journal of High Performance Computing Applications*, 28(2):129–173, 2014.

[90] Volker Springel. The cosmological simulation code gadget-2. *Monthly notices of the royal astronomical society*, 2005.

[91] Volker Springel, Naoki Yoshida, and Simon DM White. Gadget: a code for collisionless and gasdynamical cosmological simulations. *New Astronomy*, 6(2):79–117, 2001.

[92] Nitin Hemant Vaidya. *A case for multi-level distributed recovery schemes*. Texas A & M University, Computer Science Department, 1994.

[93] GD Van Albada, Bram Van Leer, and WWjun Roberts. A comparative study of computational methods in cosmic gas dynamics. pages 95–103, 1997.

[94] Rui Wang, Erlin Yao, Mingyu Chen, Guangming Tan, Pavan Balaji, and Darius Buntinas. Building algorithmically nonstop fault tolerant mpi programs. In *High Performance Computing (HiPC), 2011 18th International Conference on*, pages 1–9. IEEE, 2011.

[95] Patrick M Widener, Kurt B Ferreira, Scott Levy, and Nathan Fabian. Canaries in a coal mine: Using application-level checkpoints to detect memory failures. In *European Conference on Parallel Processing*, pages 669–681. Springer, 2015.

[96] Ren Xiaoguang, Xu Xinhai, Wang Qian, Chen Juan, Wang Miao, and Yang Xuejun. Gs-dmr: Low-overhead soft error detection scheme for stencil-based computation. *Parallel Computing*, 41:50–65, 2015.

[97] Keun Soo Yim. Characterization of impact of transient faults and detection of data corruption errors in large-scale n-body programs using graphics processing units. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 458–467. IEEE, 2014.

[98] John W. Young. A first order approximation to the optimum checkpoint interval. *Commun. ACM*, 1974.

[99] Gengbin Zheng, Xiang Ni, and Laxmikant V Kalé. A scalable double in-memory checkpoint and restart scheme towards exascale. In *Dependable Systems and Networks Workshops (DSN-W), 2012 IEEE/IFIP 42nd International Conference on*, pages 1–6. IEEE, 2012.

[100] Ziming Zheng, Andrew A Chien, and Keita Teranishi. Fault tolerance in an inner-outer solver: a gvr-enabled case study. In *International Conference on High Performance Computing for Computational Science*, pages 124–132. Springer, 2014.