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 see a World in a Grain of Sand
And a Heaven in a Wild Flower

– William Blake
## CONTENTS

LIST OF FIGURES .............................................................. vii

LIST OF TABLES .............................................................. ix

ACKNOWLEDGMENTS ............................................................ x

ABSTRACT ................................................................. xi

1 THESIS STATEMENT ........................................................... 1

2 INTRODUCTION ...................................................................... 2
   2.1 Resilience Challenge at Exascale Computing ......................... 2
   2.2 Goals and Scope of Dissertation ........................................ 6
   2.3 Contributions and Outline ............................................... 7

3 BACKGROUND AND RELATED WORK ........................................... 9
   3.1 System-level Resilience .................................................. 9
   3.2 Algorithm-level Resilience .............................................. 12
   3.3 Global View Resilience (GVR) ......................................... 14

4 APPLICATION-BASED FOCUSED RECOVERY (ABFR) ....................... 17
   4.1 Motivating Example ..................................................... 17
   4.2 Overview of ABFR Design ............................................. 19
   4.3 ABFR Operators: Encapsulate Application Recovery Knowledge .......... 22
      4.3.1 Error Check ...................................................... 23
      4.3.2 Inverse Propagation ............................................ 23
      4.3.3 Diagnosis ........................................................ 24
      4.3.4 Recovery ........................................................ 24
      4.3.5 ABFR Operator Interfaces ...................................... 25
   4.4 Runtime: Efficient, Parallel Recovery .................................. 25
      4.4.1 Basic Operator Scheduling ...................................... 26
      4.4.2 Sequential/Parallel Orchestration .............................. 26
      4.4.3 Version Management and Global View ......................... 28

5 APPLICATION STUDIES .......................................................... 29
   5.1 Stencil Computations ................................................... 29
      5.1.1 Stencil Archetype Overview .................................... 29
      5.1.2 ABFR Operators for Stencil ..................................... 30
      5.1.3 ABFR Analytical Model for Stencil ............................. 32
   5.2 N-Body Tree Computations ............................................. 42
      5.2.1 N-body Archetype Overview .................................. 42
      5.2.2 ABFR Operators for N-body .................................... 44
      5.2.3 ABFR Analytical Model for NBody Tree ..................... 46
5.3 Monte Carlo Particle Transport ................................................. 49
  5.3.1 Monte Carlo Archetype Overview ...................................... 49
  5.3.2 ABFR Operators for Monte Carlo ..................................... 49
  5.3.3 ABFR Analytical Model for Monte Carlo Particle Transport ....... 51
5.4 Discussion and Summary .................................................... 53

6 PERFORMANCE EVALUATION .................................................. 55
  6.1 Methodology ................................................................. 55
    6.1.1 Metrics ............................................................... 56
    6.1.2 Platforms ............................................................ 56
    6.1.3 Applications, Settings, and Workloads ............................ 56
  6.2 Chombo Results (Stencil) ................................................ 58
    6.2.1 Analytical Model Validation ...................................... 62
  6.3 Gadget2 Results (N-body Tree) ......................................... 65
  6.4 OpenMC Results (Monte Carlo) ........................................ 66
    6.4.1 Scale OpenMC ...................................................... 68
  6.5 Effective Recovery Focus ................................................ 72
  6.6 Discussion and Summary ................................................ 73

7 DISCUSSION AND RELATED WORK .......................................... 75
  7.1 Enable ABFR in Stencil Computations Through Compiler-based Approach .................................................. 75
    7.1.1 Approach ............................................................ 76
    7.1.2 Example of 2-D Stencil ........................................... 78
    7.1.3 Experiments ......................................................... 81
  7.2 Multiple Errors ........................................................... 83
  7.3 Related Work .............................................................. 83

8 CONCLUSIONS AND FUTURE WORK ....................................... 85
  8.1 Summary ................................................................. 85
  8.2 Future Work ............................................................... 86
LIST OF FIGURES

2.1 Latent errors: errors are detected some time after their occurrence and propagate to larger area of data space during this time period. ................................................. 3
2.2 Checkpoint-Restart approach does not work for latent errors as the last checkpoint is corrupted by the latent error. .......................................................... 4
2.3 Applicability of resilience techniques across Application knowledge and Error latency space. .......................................................... 6

3.1 Multi-level Checkpoint Restart for latent errors: iteratively restarts from checkpoints, re-executes and tests. ........................................................ 11
3.2 Distributed Arrays and Versioning on GVR .................................................. 14
3.3 GVR leverages SCR to store versions across the storages. .................................. 15
3.4 GVR performance on Cori burst buffer system. ............................................. 16

4.1 Motivating Example: Error propagated to some range of data. Recovery can be refined on error affected data instead of whole data space. .............. 18
4.2 Applications with error checking and versioning. The interval between two consecutive error checks bounds the error latency. .......................... 20
4.3 Checkpoint Restart (CR) vs. Application-based Focused Recovery (ABFR). .... 20
4.4 ABFR: Application, Operators, and Runtime .............................................. 21
4.5 Operator and other Interfaces of the ABFR Library ........................................ 25
4.6 Parallel Orchestration: potential root causes are distributed for efficient concurrent diagnosis and recovery. ............................................. 27

5.1 Stencil patterns: an error propagates to direct neighbors (blue) in a timestep. 30
5.2 Applying ABFR in a 3-point 1D Stencil. ..................................................... 31
5.3 Recovery Cost vs. Detection Interval ($M = 32768^2, t = 10^{-8}, d = 100t, r = 10^{-9}, s = 10^{-8}, \alpha = \frac{1}{4}$) .................................................. 37
5.4 Optimal Detection Interval vs. Error Rate ($M = 32768^2, p = 4096, t = 10^{-8}, r = 10^{-9}, s = 10^{-8}, \alpha = \frac{1}{4}$) .................................................. 37
5.5 Overhead vs. Error Rate Using Optimal Detection Interval ($M = 32768^2, p = 4096, t = 10^{-8}, r = 10^{-9}, s = 10^{-8}, \alpha = \frac{1}{4}$) .................................................. 39
5.6 Recovery Latency vs. Detection Interval ($M = 32768^2, m = 65536, p = 4096, n = 4, t = 10^{-8}, d = 100t, r = 10^{-9}, s = 10^{-8}, \alpha = \frac{1}{4}$) ...................... 39
5.7 Interleaved Domain Decomposition ............................................................ 40
5.8 Barnes-Hut Structures: N-Body Tree Code .............................................. 43
5.9 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. ......................... 44
5.10 Applying ABFR in a 2D N-Body Tree Computation. .................................. 45
5.11 Applying ABFR in a Monte Carlo Particle Transportation Computation. ‘X’ mark indicates particles in that batch scored to the corresponding bin. ......... 50
LIST OF TABLES

5.1 Table of Notations .......................................................... 33
5.2 Expressions for step, root, and AllRoot functions for 1, 2 and 3 dimensional grids, assuming an element interacts only with its direct neighbors. .................. 33
5.3 Summary of main notations in ABFR N-Body Tree Model. ......................... 47
5.4 Summary of main notations for ABFR MC Particle Transport model. .......... 52
5.5 Summary of ABFR operators for Stencil, N-Body Tree, and Monte Carlo computations .......................................................... 53

6.1 Operators as fraction of overall ABFR recovery: Stencil ......................... 60
6.2 Experiment Configurations ................................................... 62
6.3 Operators as fraction of overall ABFR recovery: N-Body Tree ................. 66
6.4 Operators as fraction of overall ABFR recovery: MC Neutron Transport ...... 68

7.1 ABFR Pragmas ................................................................. 77
7.2 Experiment Configuration for Varied Scales .................................. 81
7.3 Experiment Configuration for Varied Error Latencies ......................... 82
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 efficiently recover. The predicted high error rates and latent errors are a critical problem that will limit the scale of application science.

We propose a new approach called Application-Based Focused Recovery (ABFR), that defines the application knowledge needed for efficient latent error recovery. This allows the application to pursue strategies exploiting a range of application semantics within a well-defined resilience framework. The ABFR runtime then exploits this knowledge to achieve efficient latent error tolerance. ABFR enables application designers to express resilience without concern for the underlying architectures and systems. Together, these ABFR properties support flexible application-based resilience.

To demonstrate its generality, we apply ABFR 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 significant reductions in error recovery cost (up to 367x) and recovery latency (up to 24x). ABFR achieves efficient and scalable recovery at scale with high latent error rates for all three computations.
CHAPTER 1

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 generic 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 100x 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 1000-fold higher error rates in exascale systems.
CHAPTER 2
INTRODUCTION

2.1 Resilience Challenge at Exascale 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. TianHe-2 [7], Sunway Taihui Light [10]). 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 [6], Aurora [4]) will encounter obstacles [76].

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. Errors are projected to increase significantly, producing mean time between failures (MTBF) as low as a few minutes [76, 25, 50]. The error rates 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. Exascale systems must perform correctly in face of errors both from hardware and software.

Modern supercomputer hardware can detect and correct some errors [55, 68, 29], but some 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) [51, 26, 49]. Such errors are a critical concern, since their data corruption threatens the correctness of computational results. 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 real-world decisions are made based on these corrupted results, it is easy to see the necessity of ensuring correctness. It has been reported that some high-performance systems today already suffer from silent errors at a troublesome rate [75].

We focus on these latent errors, that escape system-level detection and are exposed only by sophisticated application, algorithm, and domain-semantics checks [58, 30]. Because these software checks are often computationally expensive, they are only run periodically. Figure 2.1 illustrates an example of latent error. The application data evolves from state to state. An error check is provided to examine the application data periodically. An error occurs at time $t_i$ but the error check is not invoked at that moment. Instead, at time $t_j$, the error check captures the error manifestation. The time period $t_j - t_i$ is the error detection latency, during which the error propagates through the application data space, corrupting more data.

![Figure 2.1: Latent errors: errors are detected some time after their occurrence and propagate to larger area of data space during this time period.](image)

Challenges for latent errors include determining:

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.

Figure 2.2: Checkpoint-Restart approach does not work for latent errors as the last checkpoint is corrupted by the latent error.

There are two existing approaches to tackle the problem – system-level resilience and algorithm-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. CR approach assumes immediate error detection. However,
with latent errors, single checkpoints are insufficient, as such an error could corrupt the checkpoint (see Figure 2.2). Rising error rates require frequent checkpoints for efficient execution, and fortunately new, low-cost techniques have emerged [83, 27]. 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 checkpointing does not obviously help with latent errors. Researchers have proposed multi-level checkpointing systems and multiple checkpoint-restart approaches (MCR) [60, 15, 53, 18, 66], and these systems can be applied to latent errors. When an error check fails, these systems search back through the checkpoints, restarting, reexecuting, and retesting for error. This search is expensive, so new alternatives are desirable.

Algorithm-based fault tolerance (ABFT) is a long-standing technique that exploits algorithm and data structure properties to detect and correct errors. ABFT has been extensively studied for widely used algorithms such as linear-algebra kernels [58, 73, 39, 30], including efficient schemes to correct single and double errors. ABFT can be used to address latent errors. The primary limitation 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.

System-level approaches require least effort from application programmers but adds additional cost of runtime and resources, and rarely handles latent errors. Algorithm-level approaches, on the other hand, requires application developers to support resilience as part of their application codes, including detecting error and correcting error. ABFT shows the potential to tolerate latent errors more effectively by exploiting application knowledge. These two approaches are not mutually exclusive. In fact, a general model is desired with the expectation to provide system-level support and have the capability to organize error detection and recovery for latent errors.
2.2 Goals and Scope of Dissertation

The goal of this dissertation is to introduce and demonstrate a new approach for efficient and scalable latent error recovery. We propose Application-Based Focused Recovery (ABFR), an application-system partnership for efficient latent error tolerance. ABFR bridges the gap between system support and application knowledge. 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.

Figure 2.3: Applicability of resilience techniques across Application knowledge and Error latency space.

Figure 2.3 illustrates how ABFR expands the scope of feasible application-based resilience, enabling programmers to exploit application semantics flexibly and tolerate long error latency.
error latencies. The traditional system-level approach Checkpoint-Restart is application-agnostic, and can only tolerate 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 iterative retry style recovery incurs high overhead and faces the risk of encountering more errors during recovery. In contrast, ABFT is customized for specific algorithms with the capability of recovering from medium-latency errors. It often relies on CR for storing checkpoints and recomputation, therefore fails for long-latency errors. The goal of ABFR is to provide the flexibility of exploiting application knowledge in designing resilience and equip the applications with the capability to tolerate long latent errors.

We apply ABFR to three important application archetypes (stencil, N-Body tree, and Monte Carlo particle transport) that represent important algorithmic structures and communication patterns widely-used in scientific computations. To evaluate performance benefits, we perform experiments at substantial scale, measuring performance for a range of error latencies.

2.3 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 achieve 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). 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 show large ABFR performance benefits, reducing recovery cost by 2.4x to 367x and recovery latency by 2.2x to 24x. Enable efficient application execution at extreme-scale systems with high error rates.

The remainder of the dissertation is organized as follows. Chapter 3 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 4 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 5 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 6, we present measurements from three applications that evaluate ABFR performance for varied error latencies (error rates). Discussion and related work is presented in Chapter 7, comparing ABFR with the state of arts. Finally, we summarize our work in Chapter 8, and suggest directions for future work.
CHAPTER 3  
BACKGROUND AND RELATED WORK

High error rates predicted in exascale systems threaten to jeopardize the 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 work on designing error correction techniques. We classify these approaches into two categories: system-level resilience and algorithm-level resilience (Algorithm-based Fault Tolerance, ABFT) techniques. This Chapter first describes relevant work in each category and discusses their limitations. We show that no single solution has thoroughly addresses the issue of latent errors in large-scale.

In the third section of this Chapter, we introduce Global View Resilience (GVR) library, which we use to version application data in ABFR. These intermediates will later be used to support ABFR intelligent focused recovery. GVR’s low-cost versioning enables flexible recovery for ABFR.

3.1 System-level Resilience

Checkpoint-Restart (CR) is a widely-used system-level fault tolerance technique. To avoid having to restart the application from the beginning in case of random failures, the idea of CR is to save periodically sufficient information to stable storage, such as parallel file systems, and restart the job to a previous point at which the information was saved. Such points are referred to as checkpoints, therefore the approach is called Checkpoint-Restart. 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 longer with the burden of writing checkpoints.
Many resources (storage, network bandwidth) are wasted in the error-free run. On the other hand, with less checkpoints be taken, the application runs more fast but will suffer severe work loss if a failure is presented.

The problem of tuning the checkpoint frequency is studied by many researchers. Young [82] and Daly [35] 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 $\tau_{opt}$ is given as:

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

where $\delta$ 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.

We discussed in Chapter 2 that the CR approach assumes immediate error detection. That is, the last save checkpoint is assumed correct and will be used for recovery. However, 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 [35], increasing the incidence of escaped errors. To address this reality, HPC researchers have proposed multi-level checkpointing systems and multiple checkpoint-restart (MCR) approaches [53, 18, 66]. Such systems exploit fast storage (DRAM, NVRAM) 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. 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 (see Figure 3.1). 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 high error rates.

Figure 3.1: Multi-level Checkpoint Restart for latent errors: iteratively restarts from checkpoints, re-executes and tests.

Checkpoint-Restart and Multi-level Checkpoint-Restart techniques are used to correct errors. There are also system-level supports for error detection. Replication remains the most transparent and least intrusive technique and can be used at different levels (duplication, triplication or even more). Replication is often combined with checkpointing. There are two typical types of replication schemes: process replication [50, 51] and group replication [23]. Process replication applies to message-passing applications with communicating processes. Each process is replicated, and the platform is composed of process pairs, or triplets. Group replication applies to black-box applications, whose parallel execution is replicated several times. The platform is partitioned into two halves (or three thirds). In both scenarios, results are compared before each checkpoint, which is taken only when both results (duplication) or two out of three results (triplication) coincide. If not, one or more silent errors have been detected, and the application rolls back to the last checkpoint. Not that duplication enables to detect but not to correct a latent error, while triplication enables both. Replication is not a new technique. Triple Modular Redundancy or TMR [61], is the standard fault tolerance approach for critical systems, such as embedded or aeronautical devices [16]. However, triplication has a high cost, since two thirds of the processors are executing redundant work.
To address the problem, many application-level error detection and correction techniques are therefore proposed. Algorithm-level resilience emerges as another direction to tackle the problem.

### 3.2 Algorithm-level Resilience

Algorithm-based fault tolerance (ABFR) arises as a promising alternative which can tolerate failures with low overheads. Huang and Abraham [58] 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 [73], dense matrix factorization [39], Krylov subspace iterative methods [30]. 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. [52] 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. ABFR can address long error latencies, including corrupted data across processes. Dubey et al. [40] 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.

ABFT is recognized to be more scalable [80], and an appropriate fault tolerance technique with increasing failure rate [22]. However, ABFT has only be studied for limited set of appli-
cations, 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 is an application-system approach, exploiting application knowledge for error detection, diagnose, and recovery, and supplying sophisticated runtime support to ease the programming effort, and achieving efficient recovery from latent errors.

Latent Errors Latent error problem is known as Silent Data Corruption (SDC) in high performance computing community [20, 37]. 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 [61, 65, 50], but at the cost of triple redundancy. It has been recognized that exploiting application knowledge is a much cheaper method to identify errors. Many application-specific detectors have been proposed. For example exploiting the smoothness of the evolution of a particular dataset in the iterative methods (Berrocal et al. [21]) and predicted value change (Sharma et al. [74]). 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. [20] 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 [81] explored the impact of transient faults in GPU devices on N-Body simulations and proposed utilizing physical properties to detect error.

While these efforts employed Checkpoint-Restart for recovery, ABFR can exploit them to far better effect.
3.3 Global View Resilience (GVR)

We use the GVR library [1] to preserve application data and enable flexible recovery. GVR provides a global view of array data, enabling an application to easily create, version and restore (partial or entire) arrays. In addition, GVR’s convenient naming enables 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.

GVR’s interface 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 3.2.

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 Neutron Transport, Preconditioned Conjugate Gradient solver, molecular dynamics, adaptive mesh refinement) [42, 48, 72, 84, 41]. 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 [31, 32].
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 [66] 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 3.3. 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.

(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 3.4: GVR performance on Cori burst buffer system.

We measure the GVR versioning and read performance on two burst buffer systems. NERSC Cori [5] 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 [8] is equipped with 2 TB flash memory, providing 2 GB/s bandwidth per node. Figure 3.4 presents ABFR’s performance on CORI system. And multi-versioning performance studies on JUQUEEN [8] showed GVR is able to create versions at full bandwidth, demonstrating low cost versioning is a reality [43]. In this paper, GVR’s low-cost versioning enables flexible recovery for ABFR.
CHAPTER 4
APPLICATION-BASED FOCUSED RECOVERY (ABFR)

In this chapter, we present our approach – Application-Based Focused Recovery (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 ABFR runtime system.

4.1 Motivating Example

In Chapter 2, 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 4.1. This computation evolves on a 7x7 grid with an iterative manner. Each element on the grid is updated in every timestep. An error occurred in time $t_1$ and propagated to some other data elements due to algorithmic interaction between data elements. At time $t_3$, the error is detected by some verification schemes. We want to recover the correct states of this computation. If checkpoints are provided for each timestep, we can obviously reload a previous checkpoint, for example, checkpoint at time $t_2$ and restart computation. However, this will not produce the correct results as the state in time $t_2$ is already corrupted by the error. Failed at this checkpoint-restart try, the computation can retry from checkpoint at time $t_1$ and may produce the correct results if no error occurs during the retry process.

Such iteratively rollback-recompute-verify approach works for our simple example but will quickly fail when the error rates increase or error latencies grow up. With higher error rates, the chances of encountering another error during recovery increases significantly and therefore the recovery will not succeed. In addition, Checkpoint-Restart style approach recomputes all the data, incurring tremendous computational cost.
There is one interesting observation in Figure 4.1, that only a small range of data (< 10%) is actually affected by the error. If we can accurately refine the recovery scope to this data set, the recovery cost can be reduced by 90%. The remain question is how we can identify the corrupted data. If we treat the application as blackbox, that leaves us no clue of data corruption. However, as the program designer, we master the algorithms and know how the application works. For instance, the computation has following structure.

### Algorithm 1 Example Computation Algorithm Structure

```plaintext
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 coordinates 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 verifies the state of the application, the state after verification is guaranteed to be acceptable by applications. 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 propose a framework that defines what application knowledge is needed for application designers and further provide runtime support to achieve efficient latent error recovery. In next section, we discuss the design of this framework.

4.2 Overview of ABFR 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 costly software checks required to detect errors. Figure 4.2 illustrates a typical execution model of application equipped with error checking and versioning.

When a latent error is detected, recovery can be safely done from a correct copy of application state, captured at the last successful latent-error check. Restarting the application and recomputing from that correct copy yields a checkpoint-restart (CR) style recovery for latent errors as shown in Figure 4.3a. The two problems with the CR-style approach is that
Figure 4.2: Applications with error checking and versioning. The interval between two consecutive error checks bounds the error latency.

Figure 4.3: Checkpoint Restart (CR) vs. Application-based Focused Recovery (ABFR).

Recomputation involves the full application and for the maximum potential error latency – the error checking period.

Application-based focused recovery (ABFR) improves latent error resilience by exploiting application knowledge to reduce the recovery effort. As shown in Figure 4.3b, application knowledge often enables 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. However, the idea of exploiting application knowledge to reduce resilience cost is not novel[58], 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 4.4: ABFR: Application, Operators, and Runtime](image)

In Figure 4.4, 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, but with 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 5, 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 four basic operators, then introduce the ABFR runtime design.

4.3 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 4.2). 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 check on application-level is usually expensive. First, frequently checking application states contradict the purpose of efficient and scalable application execution. Second, many applications inherently save intermediate results for future reference. At last, these intermediate states can help refining recovery scope in case of an latent error. It is recognized that versioning also incurs overhead. The question remained to solve is how to determine the frequency of error checking and versioning. As with Young and Daly’s [82, 35] 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 5, we will discuss detailed design for three exemplar scientific applications. At last, we present the ABFR operator interfaces.

### 4.3.1 Error Check

**Definition:** If it fails, the error check returns the location and value of program data structure elements with corrupted values. Error checks are assumed to be expensive.  

To address the problem of latent (or silent) errors, many application-specific error checks have been proposed. Much effort [58, 22, 73] 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.

### 4.3.2 Inverse Propagation

**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 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

---

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.
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 but may not be useful 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 5 we discuss the design of inverse propagation for three different communication patterns.

4.3.3 Diagnosis

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 are still a large set. To bound error impact more precisely, PRCs can be tested (diagnosis) exploiting deep application knowledge, eliminating many of the initial PRCs. 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. PRCs in different versions can be distributed to available tasks and tested concurrently, therefore further reducing recovery latency.

4.3.4 Recovery

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 recovery [59, 31, 62] and approximate recovery [33].
4.3.5 ABFR Operator Interfaces

The operator interfaces are presented in Figure 4.5. 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 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.

4.4 Runtime: Efficient, Parallel Recovery

ABFR provides a clear separation between application knowledge and 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 [32].

4.4.1 Basic Operator Scheduling

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.

4.4.2 Sequential/Parallel Orchestration.

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 4.6. We describe several examples.
Figure 4.6: Parallel Orchestration: potential root causes are distributed for efficient concurrent diagnosis and recovery.

- **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 [52].

- **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 [52], 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.

4.4.3 Version Management and Global View

Versioning of application data is accomplished by using GVR (see Chapter 3 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 6.

The example in Figure 4.6 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. Consequently, the recovery work is not balanced. Instead of reloading all the versions, ABFR is able to only materialize 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.
CHAPTER 5
APPLICATION STUDIES

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 US DOE’s NERSC [9] 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.

5.1 Stencil Computations

5.1.1 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 [36, 45], (e.g. cosmology, combustion and image processing). Stencil codes iterate on an array:

Figure 5.1 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 [67].
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

Figure 5.1: Stencil patterns: an error propagates to direct neighbors (blue) in a timestep.

5.1.2 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 5.2b). We use this operator for experiments in Chapter 6. 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
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 5.2d). 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) [21]. 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 [20, 74, 40], 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 5.2. 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.

### 5.1.3 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 $\text{step}(j)$ 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
<table>
<thead>
<tr>
<th>Variable</th>
<th>Definitions</th>
<th>Units</th>
</tr>
</thead>
<tbody>
<tr>
<td>$M$</td>
<td>Application size (number of elements × element size)</td>
<td>bytes</td>
</tr>
<tr>
<td>$m$</td>
<td>Box Size (number of elements in one box × element size)</td>
<td>bytes</td>
</tr>
<tr>
<td>$n$</td>
<td>Number of boxes assigned to one process</td>
<td>-</td>
</tr>
<tr>
<td>$p$</td>
<td>Number of processes in computation</td>
<td>-</td>
</tr>
<tr>
<td>$t$</td>
<td>Time to advance one element by one timestep</td>
<td>seconds/byte</td>
</tr>
<tr>
<td>$d$</td>
<td>Time to run the detector on one element</td>
<td>seconds/byte</td>
</tr>
<tr>
<td>$s$</td>
<td>Time to store one element (versioning)</td>
<td>seconds/byte</td>
</tr>
<tr>
<td>$r$</td>
<td>Time to reload one element</td>
<td>seconds/byte</td>
</tr>
<tr>
<td>$c$</td>
<td>Time to compare one element with a previous version</td>
<td>seconds/byte</td>
</tr>
<tr>
<td>$D$</td>
<td>Detection interval, Error Latency Bound</td>
<td>timesteps</td>
</tr>
<tr>
<td>$V$</td>
<td>Versioning interval</td>
<td>timesteps</td>
</tr>
<tr>
<td>$α$</td>
<td>Ratio of versioning interval to detection interval, $V = αD$</td>
<td>-</td>
</tr>
<tr>
<td>$B$</td>
<td>Number of versions between two detections, $B = \frac{D}{V} = \frac{1}{α}$</td>
<td>-</td>
</tr>
<tr>
<td>$λ$</td>
<td>Error rate</td>
<td>errors/(second*byte)</td>
</tr>
<tr>
<td>$λM$</td>
<td>System error rate</td>
<td>errors/second</td>
</tr>
<tr>
<td>$(1 - e^{-λM})$</td>
<td>Probability of having an error in one second</td>
<td>-</td>
</tr>
<tr>
<td>$E$</td>
<td>Expected cost of completing computation of $D$ timesteps</td>
<td>(cpu) seconds</td>
</tr>
<tr>
<td>$Rec$</td>
<td>Recovery cost: the amount of work required to recover</td>
<td>(cpu) seconds</td>
</tr>
<tr>
<td>$T$</td>
<td>Expected runtime of completing computation of $D$ timesteps</td>
<td>seconds</td>
</tr>
<tr>
<td>$RecLat$</td>
<td>Recovery latency: runtime critical path for recovery</td>
<td>seconds</td>
</tr>
</tbody>
</table>

Table 5.1: Table of Notations

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:

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

<table>
<thead>
<tr>
<th>$step(i)$</th>
<th>1D</th>
<th>2D</th>
<th>3D</th>
</tr>
</thead>
<tbody>
<tr>
<td>$root(i)$</td>
<td>2i + 1</td>
<td>4i</td>
<td>$4i^2 + 2$</td>
</tr>
<tr>
<td>$AllRoot$</td>
<td>$D^2$</td>
<td>$\frac{2}{3}D^3 + \frac{1}{3}D$</td>
<td>$\frac{1}{3}D^4 + \frac{2}{3}D^2$</td>
</tr>
</tbody>
</table>

Table 5.2: Expressions for $step$, $root$, and $AllRoot$ functions for 1D, 2D, and 3D stencils, assuming an element interacts only with its direct neighbors.

Table 5.2 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 recom-
puted 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 5.2c:

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

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 5.2d:

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

As discussed in Chapter 4, 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 [13] for the analysis of 1D and 3D stencils.

**Recovery Cost**  Let \( E_{ABFR} \) 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 \( \lambda \) (errors per second per byte) and detection interval \( D \). In this section, we compare it with the classical CR (Checkpoint/Restart) approach, denoted by \( E_{CR} \).

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 $\lambda M$ is the application error rate. Therefore, we can write $E_{CR}$ and $E_{ABFR}$ as functions of $D$ and $\lambda M$ as follows:

$$E_{CR} = DtM + dM + sM + \left(1 - e^{-\lambda M \frac{DtM}{p}}\right) Rec_{CR},$$ \hspace{1cm} (5.1)

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

(5.2)

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:

$$Rec_{CR} = rM + DtM.$$

(5.3)

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 $jV$:

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

Therefore, $\frac{A(j)}{All\text{Root}}$ denotes the probability that the error occurred between version $j$ and $j + 1$, and we can write:

$$Rec_{ABFR} = \sum_{j=0}^{B-1} \frac{A(j)}{All\text{Root}} (diag(j) + recomp(j)).$$
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 $(r + c)\text{root}(kV)$ for every version $k$ that passed the diagnosis test, that is from version $B - 1$ to $j$ included. Therefore, we can write:

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

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 $\text{root}((j + 1)V - 1)$ potential root causes elements to recompute. At every timestep, the number of elements to recompute increases by $\text{step}(j)$, so that there are a total of $\text{root}(2(j + 1)V)$ elements to recompute at timestep 0. Therefore, we can write:

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

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 $D$:

$$
\text{Rec}_{ABFR} = \frac{8}{15} t(\alpha^5 - 5\alpha^3 + 9\alpha + 5)D^3 + O(D^2),
$$

where $\alpha = \frac{1}{B}$. 

**Recovery Cost Comparison** The dominant cost in recovery is recomputation. It is $O(DM)$ for CR in Equation 5.3 and $O(D^3)$ for ABFR in Equation 5.4. Suppose the number 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 constrast, ABFR only needs to recompute a small fraction of the $M$ elements. The corresponding recomputation cost is
Figure 5.3: Recovery Cost vs. Detection Interval ($M = 32768^2, t = 10^{-8}, d = 100t, r = 10^{-9}, s = 10^{-8}, \alpha = \frac{1}{4}$)

Figure 5.4: Optimal Detection Interval vs. Error Rate ($M = 32768^2, p = 4096, t = 10^{-8}, r = 10^{-9}, s = 10^{-8}, \alpha = \frac{1}{4}$)

$O(D^2)$, $O(D^3)$ and $O(D^4)$ respectively (see [13]). 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 5.3 (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 $H = \frac{E}{DLM}$ denote the expected overhead with respect to the computation cost without errors. Using Taylor series to approximate $1 - e^{-\lambda M \frac{DLM}{p}}$ to $\lambda M \frac{DLM}{p}$ (up to first-order
terms), we obtain:

\[ H_{CR} = 1 + \frac{d + s}{Dt} + \frac{\lambda M}{p} (rM + DtM), \]
\[ H_{ABFR} = 1 + \frac{b}{D} + \frac{\lambda M}{p} aD^3, \]  
(5.5)

where \( a = \frac{8}{15} t(\alpha^5 - 5\alpha^3 + 9\alpha + 5) \) and \( b = \frac{\alpha d + s}{\alpha t} \).

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

\[ D^*_{CR} = \sqrt{\frac{(d + s)p}{\lambda M^2 t^2}}, \]  
and \[ D^*_{ABFR} = \frac{4b}{3a\lambda M}. \]  
(5.6)

Empirical studies of petascale systems have shown MTBF’s of three hours at deployment [38], and allowing for the greater scale of exascale systems [76, 27], future HPC system MTBFs have been projected as low as 20 minutes [50]. 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 \( \lambda M \) to the system error rate.

We plot the optimal detection interval as a function of the error rate \( \lambda M \) in Figure 5.4. 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

\[ H_{CR}^* = 1 + 2M \sqrt{\frac{(d + s)p}{\lambda M^2 t^2}} + rM^2 \lambda, \]  
(5.7)
\[ H_{ABFR}^* = 1 + \frac{4}{3} \sqrt{\frac{3a^3 \lambda M}{p}}. \]  
(5.8)
We plot the overhead as a function of error rate, when using the optimal detection interval, in Figure 5.5. With growing error rates, CR incurs high overhead. In contrast, ABFR significantly reduces overhead and performs stably even for high error rates.

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 5.7, 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.

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:

$$RecLat_{CR} = n(rm + Dtm).$$ (5.9)
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 $n_{\text{ideal}} = \frac{\text{root}(D)}{mp} = \frac{2D^2}{mp} + O(D)$. For the interleaved mapping (see Figure 5.7), 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 $n_{\text{inter}} = \frac{2D}{\sqrt{m}} \sqrt{p} = \frac{2D\sqrt{M}}{mp}$ corrupted boxes. Then, for an error at step $j$, we have:

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

$$\text{recomp}(j) = t \sum_{k=0}^{(j+1)V} m + s \sum_{k=0}^{j+1} m.$$
To compute the recovery latency $\text{Rec}_{\text{box}}$ per box, we proceed as before:

$$
\text{Rec}_{\text{box}} = B - 1 \sum_{j=0}^{B-1} \frac{A(j)}{\text{AllRoot}} (\text{diag}(j) + \text{recomp}(j))
$$

$$
= t m \alpha D + o(D).
$$

Multiplying $\text{Rec}_{\text{box}}$ by the corresponding number of boxes in the ideal and interleaved scenarios, we obtain

$$
\text{RecLat}_{\text{ideal}} = \frac{2t \alpha}{p} D^3 + O(D^2), \quad (5.10)
$$

$$
\text{RecLat}_{\text{inter}} = \frac{2t \alpha \sqrt{M}}{p} D^2 + O(D). \quad (5.11)
$$

Comparing Equations (5.9) and (5.10), 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 $\text{RecLat}_{\text{CR}}$ and $\text{RecLat}_{\text{inter}}$ as a function of detection interval in Figure 5.6. Similar as in Figure 5.3, 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.

$$
T_{CR} = D n m t + d n m + s n m + (1 - e^{-\lambda M D n m t}) \text{RecLat}_{CR}
$$

$$
T_{ABFR} = D n m t + d n m + B s n m + (1 - e^{-\lambda M D n m t}) \text{RecLat}_{ABFR}
$$
The overhead $H = \frac{T}{D_{nm}}$ of CR and ABFR are given by

\[
H_{CR} = 1 + \frac{d + s}{Dt} + \lambda M n (r m + D t) , \\
H_{ideal} = 1 + \frac{\alpha d + s}{\alpha D t} + \lambda M \frac{2 t \alpha}{p} D ^ 3 , \\
H_{inter} = 1 + \frac{\alpha d + s}{\alpha D t} + \lambda M \frac{2 t \alpha \sqrt{M}}{p} D ^ 2 . 
\]

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

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

The optimal interval $D_{CR}^*$ of CR is the same as in Equation (5.6). The optimal interval for ideal-ABFR is $D_{ideal}^* = \Theta(\lambda^{-\frac{1}{2}})$, the same order of magnitude as $D_{ABFR}^*$, the optimal value of Equation (5.6) for the recovery cost. $D_{inter}^*$ is different due to imbalanced recovery.

## 5.2 N-Body Tree Computations

### 5.2.1 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 [11, 12].

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 [44, 79, 14, 54]. We select Barnes-Hut [17] as an exemplar of this class of tree codes.
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 5.8). 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. $\theta$ 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 against force computation and particle movement time for best efficiency [64]. We adopt a dynamic update scheme where tree nodes can be updated without reconstructing the entire tree at every timestep [78]. Algorithm 3 shows the pseudocode of N-Body tree computations.

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 contami-
Figure 5.9: 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.

Algorithm 3  N-Body Tree Computation Algorithm Structure

\[
\text{for } K \text{ timesteps do} \\
\quad \text{Construct Barnes-Hut tree, update internal nodes.} \\
\quad \text{for } k_{local} \text{ timesteps do} \\
\quad\quad \text{for } N \text{ particles do} \\
\quad\quad\quad \text{Walk tree (Open or Not) and compute forces} \\
\quad\quad \text{end for} \\
\quad \text{end for} \\
\text{end for}
\]

nated). 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 [56]. 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 5.9). Thus our ABFR approach detects when errors are clustered, and focuses diagnosis to reduce error recovery cost.

5.2.2 ABFR Operators for N-body

Inverse Propagation. Given an error manifestation – erroneous particle – the tree struc-
Figure 5.10: Applying ABFR in a 2D N-Body Tree Computation.
ture 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 5.10b. 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 5.10c). 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 [56, 81], 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 5.10. 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.

### 5.2.3 ABFR Analytical Model for NBody 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
Definitions

<table>
<thead>
<tr>
<th>Listings</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$H$</td>
<td>Height of tree</td>
</tr>
<tr>
<td>$N$</td>
<td>Number of particles</td>
</tr>
</tbody>
</table>

Error Rate

<table>
<thead>
<tr>
<th>Listings</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\lambda$</td>
<td>Errors per second per particle</td>
</tr>
</tbody>
</table>

Time

<table>
<thead>
<tr>
<th>Listings</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c$</td>
<td>Time to compute one leaf</td>
</tr>
<tr>
<td>$d$</td>
<td>Time to detect errors on one leaf</td>
</tr>
<tr>
<td>$v$</td>
<td>Time to version one leaf</td>
</tr>
<tr>
<td>$r$</td>
<td>Time to reload data of one leaf</td>
</tr>
</tbody>
</table>

Tree-wise

<table>
<thead>
<tr>
<th>Listings</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_c$</td>
<td>Time to compute the tree without errors</td>
</tr>
<tr>
<td>$T_d$</td>
<td>Time for detection the tree without errors</td>
</tr>
<tr>
<td>$T_v$</td>
<td>Time for versioning the tree without errors</td>
</tr>
</tbody>
</table>

Frequency

<table>
<thead>
<tr>
<th>Listings</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L$</td>
<td>Latency Bound (Detection interval) timesteps</td>
</tr>
<tr>
<td>$V$</td>
<td>Versioning interval (timesteps)</td>
</tr>
</tbody>
</table>

Functions

<table>
<thead>
<tr>
<th>Listings</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f(h)$</td>
<td>Opening rate at tree height $h$</td>
</tr>
</tbody>
</table>

Table 5.3: Summary of main notations in ABFR N-Body Tree Model.

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 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 $\frac{L}{T}$ 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.

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

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

$$
= T_c + N \cdot (d + v) + (1 - e^{-\lambda T_c})((r + c) \cdot f(h) \cdot \frac{L}{2V} + \sum_{i=1}^{\frac{L}{2V}} (f(h) \cdot N)^i \cdot c)
$$

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.
5.3 Monte Carlo Particle Transport

5.3.1 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 [19, 24, 63]. 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.

5.3.2 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) 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 5.11b). Using this bit-vector
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

Figure 5.11: Applying ABFR in a Monte Carlo Particle Transportation Computation. ‘X’ mark indicates particles in that batch scored to the corresponding bin.

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 5.11c. 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 5.11. 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.

### 5.3.3 ABFR Analytical Model for Monte Carlo Particle Transport

**Expected Cost of Checkpoint-Restart** Let $E(T_{CR})$ denote the expected total cost of Checkpoint-Restart approach. It includes computing, versioning, detection and recovery.
<table>
<thead>
<tr>
<th>N</th>
<th>Number of particles per batch</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>Number of batches</td>
</tr>
<tr>
<td>S</td>
<td>Tally size</td>
</tr>
<tr>
<td>L</td>
<td>Latency Bound (Detection interval), batches</td>
</tr>
<tr>
<td>V</td>
<td>Versioning interval (batches)</td>
</tr>
<tr>
<td>c</td>
<td>Time to compute one particle</td>
</tr>
<tr>
<td>d</td>
<td>Time to detect one batch</td>
</tr>
<tr>
<td>v</td>
<td>Time to version one batch</td>
</tr>
</tbody>
</table>

Functions

<table>
<thead>
<tr>
<th>$f_{batch}$</th>
<th>Contribution factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f_{particle}$</td>
<td>Contribution factor</td>
</tr>
</tbody>
</table>

Table 5.4: Summary of main notations for ABFR MC Particle Transport model.

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

**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 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

\[
\mathbb{E}(T_{ABFR}) = T_c + B \cdot d + v + (1 - e^{-\lambda T_c}) T_{rcmp}
\]

\[ = T_c + B \cdot d + v + (1 - e^{-\lambda T_c})(B \cdot f_{batch} \cdot N \cdot f_{particle} \cdot c). \]

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.

### 5.4 Discussion and Summary

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

Table 5.5: 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. 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. 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 [59, 62] and approximate recovery [33] could also be used. Table 5.5 compares the design of four operators for three computations.
CHAPTER 6
PERFORMANCE EVALUATION

6.1 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 5, and compare to a latent-error recovery scheme based on Checkpoint-Restart (CR), as described in Chapter 4.

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 [32], 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 [82, 35]. In other work, we built an analytical models for ABFR that can determine the optimal intervals [47, 28].

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 neutron 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.
6.1.1 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 4, 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.

6.1.2 Platforms

Experiments were conducted on two platforms.

- **Midway2.** Shared compute cluster at the University of Chicago [3], 382 nodes, linked by non-blocking FDR/EDR Infiniband. Nodes: 28-core, 2.4Ghz Intel Broadwell with 64 GB memory.

- **Edison.** DOE supercomputer [2], 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.

6.1.3 Applications, Settings, and Workloads

**Chombo Heat Equation.** Chombo [34] 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 [77] 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 [71] 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 [57], 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 [70]. 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.
6.2 Chombo Results (Stencil)

In Figure 6.1a, 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.
Figure 6.2: Recovery Latency: ABFR vs. CR, various error latency bounds.

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 6.3 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 [47]. 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.

<table>
<thead>
<tr>
<th>Latency Bound (timesteps)</th>
<th>Inverse Propagation</th>
<th>Diagnosis</th>
<th>Recovery</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,000</td>
<td>0.04%</td>
<td>56.1%</td>
<td>43.8%</td>
</tr>
<tr>
<td>5,000</td>
<td>0.0%</td>
<td>77.1%</td>
<td>22.9%</td>
</tr>
<tr>
<td>9,000</td>
<td>0.0%</td>
<td>83.6%</td>
<td>16.4%</td>
</tr>
<tr>
<td>13,000</td>
<td>0.0%</td>
<td>75.6%</td>
<td>24.4%</td>
</tr>
<tr>
<td>Recovery Cost</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1,000</td>
<td>0.0%</td>
<td>56.1%</td>
<td>43.9%</td>
</tr>
<tr>
<td>5,000</td>
<td>0.0%</td>
<td>62.8%</td>
<td>37.2%</td>
</tr>
<tr>
<td>9,000</td>
<td>0.0%</td>
<td>62.9%</td>
<td>37.1%</td>
</tr>
<tr>
<td>13,000</td>
<td>0.0%</td>
<td>54.1%</td>
<td>45.9%</td>
</tr>
</tbody>
</table>

Table 6.1: Operators as fraction of overall ABFR recovery: Stencil

Figure 6.2a 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 reduc-
tion 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 6.1 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 6.4, 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 6.4: Stencil: Parallel Diagnosis vs. Sequential Diagnosis](image)

Figure 6.4: Stencil: Parallel Diagnosis vs. Sequential Diagnosis
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 6.2. 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

**Recovery Cost** Figure 6.5 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

<table>
<thead>
<tr>
<th>Number of ranks</th>
<th>4096</th>
</tr>
</thead>
<tbody>
<tr>
<td>Domain size</td>
<td>$10^9$ (32768x32768)</td>
</tr>
<tr>
<td>Number of boxes</td>
<td>16384 (128x128)</td>
</tr>
<tr>
<td>Box size</td>
<td>65536 (256x256)</td>
</tr>
<tr>
<td>#Box per process</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 6.2: Experiment Configurations

### 6.2.1 Analytical Model Validation
Figure 6.5: Recovery Cost vs. Detection Interval (Model plotted for experiment configuration and measured $t = 1.5 \times 10^{-8}$ second)

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 6.5 also plots the performance model (dotted and dashed lines), showing a close match (for broader comparison see Figure 5.3). As expected, ABFR cost starts lower and grows polynomially with the detection interval.

**Recovery Latency** Figure 6.6 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 6.6 are performance model results using parameter
Figure 6.6: Recovery Latency vs. Detection Interval (Model plotted for experiment configuration and measured $t = 1.5 \times 10^{-8}$ second)

Figure 6.7: Data Read (MB) vs. Detection Interval

values of our experiments (see also Figure 5.6). 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
Data Read (IO) An important cost for recovery is the reading of stored version data from the IO system. Figure 6.7 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.

6.3 Gadget2 Results (N-body Tree)

Figure 6.1b 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 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 6.3). The recovery operator recomputes the remaining PRCs for the latency, so its cost grows linearly with error latency bound.

Figure 6.2b 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.

Table 6.3 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 computa-
<table>
<thead>
<tr>
<th>Latency Bound (timesteps)</th>
<th>Recovery Cost</th>
<th>Inverse Propagation</th>
<th>Diagnosis</th>
<th>Recovery</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>0.0%</td>
<td>0.8%</td>
<td>99.2%</td>
<td></td>
</tr>
<tr>
<td>200</td>
<td>0.0%</td>
<td>0.7%</td>
<td>99.3%</td>
<td></td>
</tr>
<tr>
<td>300</td>
<td>0.0%</td>
<td>0.7%</td>
<td>99.3%</td>
<td></td>
</tr>
<tr>
<td>Recovery Latency</td>
<td>100</td>
<td>0.0%</td>
<td>1.2%</td>
<td>98.8%</td>
</tr>
<tr>
<td></td>
<td>200</td>
<td>0.0%</td>
<td>0.9%</td>
<td>99.1%</td>
</tr>
<tr>
<td></td>
<td>300</td>
<td>0.0%</td>
<td>1.0%</td>
<td>99.0%</td>
</tr>
</tbody>
</table>

Table 6.3: Operators as fraction of overall ABFR recovery: N-Body Tree

The inverse propagation and diagnosis operators exploit algorithms and data structure properties, and are demonstrated to be lightweight.

### 6.4 OpenMC Results (Monte Carlo)

Figure 6.1c 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 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 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.

For recovery latency, see Figure 6.2c, 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
Figure 6.8: Achieved Recovery Focus: Operator impact on recovery work

less than that for recovery cost.

Table 6.4 compares the cost of each operator in Monte Carlo Neutron Transport computation. Distinguished from stencil and N-Body tree’s lightweight inverse propagation operator,
MC Neutron Transport has a relative costly inverse propagation operator. Adding the bit-vecto
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 6.3 shows the cost fraction of each operator. The inverse propagatio
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.

### 6.4.1 Scale 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 6.9: Strong scaling, recovery cost: ABFR vs. CR

Figure 6.10: Strong scaling, recovery latency: ABFR vs. CR

Figure 6.9 presents the recovery cost of ABFR, and Figure 6.9 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
Figure 6.11: Strong scaling, amount of recomputation needed after each operator 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 6.11. The amount of recomputation needed 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 6.12 compares the recovery cost of ABFR and CR, and Figure 6.12 compares the recovery latency of ABFR and CR for error latency bounds from 10 to 30 batches. ABFR 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 number of events to be scored become more frequent, increasing the density of tally arrays.
Figure 6.14: Weak scaling, amount of recomputation needed after each operator 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.

### 6.5 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 6.8 shows how
the required recomputation shrinks with each operator. After error check, the needed 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 6.8a) inverse propagation reduces the amount of recomputation required 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 6.8b), 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.

6.6 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% [32]. 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 [47].

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) [31], and more samples may suffice to meet the convergence condition.
CHAPTER 7
DISCUSSION AND RELATED WORK

In this section, we first discuss an approach to easily support ABFR in applications. ABFR defines the application knowledge needed for efficient latent error recovery. 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. In previous Chapters, we only examine the single error in modeling and experiments. We also discuss how to apply ABFR for multiple errors. We further compare ABFR with the state of arts, discussing the difference and advantages of ABFR.

7.1 Enable ABFR in Stencil Computations Through Compiler-based Approach

We present a compiler-based approach to automatically add ABFR functions and GVR versioning functions [32] to perform ABFR resilience for stencil computations in a simplified and portable way. ¹

We implement a translator using the ROSE source-to-source compiler infrastructure [69]. 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 executables from the transformed code. The library is released in [46], 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 re-

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

7.1.1 Approach

ABFR defines a generic framework for application-based recovery: four basic operators (error check, inverse propagation, diagnosis and recovery) are implemented by developers using application knowledge (refer to [47] 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.

```c
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 7.1: 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 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
Table 7.1: ABFR Pragmas

<table>
<thead>
<tr>
<th>Pragma</th>
<th>Attributes</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>#pragma abfr_gvr_init argc argv</td>
<td>argc: the number of command line arguments, argv: command line arguments</td>
<td>Initialize GVR and ABFR. Called forefront.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>#pragma abfr_gvr_finalize</td>
<td>-</td>
<td>Finalize GVR and ABFR in the end.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>#pragma abfr_init_version data ndims count type gds</td>
<td>data: data to protect, ndims: dimension of stencil, count: number of elements in each dimension, type: data type, gds: name for GVR handler.</td>
<td>Initialize GVR versions for data backup</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>#pragma abfr_versioning data ld lo_index hi_index gds</td>
<td>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,</td>
<td>Create a version (snapshot) of data</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>#pragma abfr_recover data gds error_index step latency interval error_tol invert_propagation compute</td>
<td>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</td>
<td>Invoke ABFR recovery function</td>
</tr>
</tbody>
</table>
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 an 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 7.1. ROSE-ABFR translator parses pragmas and adds ABFR/GVR functions respectively.

### 7.1.2 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 7.2. Figure 7.3 shows the output of the translator, that a GVR initialize 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 7.2: pragma abfr_gvr_init

We add **abfr_gvr_finalize** in the end of main function as shown in Figure 7.4. Figure 7.5 shows the output. GVR finalize function is called after pragma.
To allocate GVR array, we specify the data structure to protect, i.e. \( u \), the number of dimension of stencil in \( \text{ndims} \) variable and the size of each dimension in \( \text{count} \) variable, the data type \text{double} and a user-define GVR handler \text{gds\_u} as shown in Figure 7.6. Figure 7.7 illustrates the added code for allocating GVR array.

![Figure 7.3: Output of ROSE-ABFR translator for pragma abfr\_gvr\_init](image)

![Figure 7.4: pragma abfr\_gvr\_finalize](image)

![Figure 7.5: Output of ROSE-ABFR translator for pragma abfr\_gvr\_finalize](image)

![Figure 7.6: pragma abfr\_init\_version](image)

![Figure 7.7: Output of ROSE-ABFR translator for pragma abfr\_init\_version](image)

![Figure 7.8: Output of ROSE-ABFR translator for pragma abfr\_init\_version](image)

![Figure 7.9: Output of translator is shown in Figure 7.9. It adds](image)
GVR put and versioning function calls.

```c
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 7.8: pragma abfr_versioning

```c
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 7.9: Output of ROSE-ABFR translator for pragma abfr_versioning

Figure 7.10 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 7.11.

```c
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 7.10: pragma abfr_recover

```c
238 #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
241   u);
```

Figure 7.11: Output of ROSE-ABFR translator for pragma abfr_recover

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

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 7.2.

Table 7.2: Experiment Configuration for Varied Scales

<table>
<thead>
<tr>
<th></th>
<th>128, 256, 512, 1024</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of processes</td>
<td>128, 256, 512, 1024</td>
</tr>
<tr>
<td>Stencil size per process</td>
<td>8192 * 8192</td>
</tr>
<tr>
<td>Total timesteps</td>
<td>10,000</td>
</tr>
<tr>
<td>Error latency</td>
<td>1000</td>
</tr>
</tbody>
</table>

Figure 7.12 shows the recovery cost for varied scales. The error free runtime is around 47 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.

Second, we explore the recovery performance for varied error latencies. Experiment configuration is listed in Table 7.3.

Figure 7.13 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.

Table 7.3: Experiment Configuration for Varied Error Latencies

<table>
<thead>
<tr>
<th>Number of processes</th>
<th>1024</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stencil size per process</td>
<td>8192 * 8192</td>
</tr>
<tr>
<td>Error latency</td>
<td>500, 1000, 1500, 2000, 2500, 3000, 3500, 4000</td>
</tr>
</tbody>
</table>

Figure 7.13: Recovery overhead vs. Error Latency
7.2 Multiple Errors

For simplicity we only experimented 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.

7.3 Related Work

In Chapter 3, we present two directions to solve the resilience problem: system-level resilience and algorithm-level resilience.

We have compared the performance of ABFR with CR in Chapter 6. In all studies, the cost of CR grows linearly with the error latency and application scale. With growing scale and error rates, CR will incur high overheads. Note that, the Checkpoint-Restart approach employed in our study for comparison is an improved version of classical CR because it avoids iterative rollback and recompute until the error is corrected. The CR scheme only saves a version in memory after each error check, avoiding heavy I/O cost enforced by some checkpointing schemes. We didn’t compare the performance with Multi-level Checkpoint-Restart but the performance of MCR is bounded by the results of CR measured in our experiments. Because MCR relies on iterative search, retry, and test, repeating the procedure of the improved CR in this thesis. ABFR solves the scalability
problem of CR by exploiting application knowledge to focus recovery on where needed. The cost of ABFR is independent of the application scale and problem size. Instead it is more determined by the error latency and application semantics. Although the quality of operators are guaranteed by the developers, they can easily experiment with different strategies and have a range of choices for their design.

Algorithm-level resilience is more promising and demonstrated to incur low recovery overhead for some applications. However, there are no general principles or models for applying ABFT for all the applications. In contrast, ABFR provides a generic 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. With the idea of inverse propagation and diagnosis, the recovery scope can further refined. Hence, we believe ABFR is more general and achieves better latent error recovery performance compared to the state of art ABFT techniques.
CHAPTER 8
CONCLUSIONS AND FUTURE WORK

8.1 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 thesis, I presented Application-Based Focused Recovery (ABFR), a new approach that 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 thesis described the ABFR operator definition and interfaces, providing a general framework on how to exploit application knowledge for latent error resilience. I presented an ABFR runtime that intelligently exploits parallelism and 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.

Application of three diverse archetypes demonstrates the breadth of ABFR’s applicability, and illustrates both what is involved in ABFR operators, and the numerous opportunities for flexible ABFR for latent errors. 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 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.

Experiments with application codes (Chombo heat equation, Gadget2, OpenMC) demonstrate the scalability and efficiency of ABFR runtime and overall ABFR recovery. Specifically, ABFR reduces recovery cost by 2.4x to 367x and recovery latency by 2.2x to 24x compared to Checkpoint-Restart approach, demonstrating ABFR achieves scalable recovery.
performance on large-scale systems.

Overall, the thesis demonstrated that ABFR is an efficient and scalable application-based resilience approach for large-scale systems.

8.2 Future Work

Future directions involve increasing the breadth of experience with ABFR, creation of analytic models to select intervals, and consideration of multiple latent error scenarios. 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. With ABFR’s dramatically higher efficiency in addressing latent error, how does this balance change? Can a unified model to select optimal error checking and versioning intervals for latent errors be created? Another interesting direction is to support reuse of ABFR functions, further reducing programmer effort. ABFR defines the application knowledge needed for efficient latent error recovery. 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. We experimented with a compiler-based approach to automatically add ABFR and GVR functions in stencil computations and demonstrated efficient and scalable recovery. We will examine other possibilities across a range of applications. Finally, our application studies assume single latent errors (a single root cause), but nothing in the ABFR approach limits it to single errors. Broad study of multiple-latent error recovery operators for ABFR is indeed an interesting research direction.
BIBLIOGRAPHY


cost of error recovery in resilient amr scientific applications. In Proceedings of the
International Conference for High Performance Computing, Networking, Storage and

application structure aware resilience through differentiated state saving and recovery.

monte carlo neutron transport simulations using global view arrays. The International

performance opportunities in bgas system for resilience. In International Conference on


Sons, 2013.


with latent errors. In Parallel Processing (ICPP), 2017 46th International Conference


