ABSTRACT

KHETAWAT, HARSH. Pragma-Based Compiler Extension for End-to-End Resiliency Against Soft Faults. (Under the direction of Dr. Rainer Frank Mueller.)

Widely varying techniques have been investigated in order to protect applications from faults. A number of resilience techniques exist such as checkpoint/restart, redundant computation, algorithm-based fault tolerance, and so on. While one technique may impose the least overhead and best protection, another kernel may call for a different resilience technique. In the future this might result in programs which have a sequence of kernels each protected by a disparate resilience mechanism. Combining such techniques are often non-trivial and might lead to applications and data-structures that are vulnerable to soft faults going uncorrected and worse still, undetected, particularly for variables live across multiple kernels.

This work aims to design, implement, and evaluate a pragma-based compiler extension to combine various resiliency techniques in a manner that reduces the vulnerability across kernels. The pragma-based directive allows application programmers to focus on algorithms and performance while lifting the burden of combining resiliency techniques in an end-to-end manner and the challenges with data protection that it may cause. By expending minimal programming effort, the application programmer can specify a per-kernel pragma that not only protects the data structure from corruption within the kernel but also across kernels, extending through the entire life-time of the data structure.

In order to demonstrate the effectiveness of the compiler extension, we evaluate two applications, sequentially composed matrix multiplication and TF-IDF, by injecting faults at different rates into the data structure and measuring the resulting overhead. The experimental results show that our technique successfully detects and corrects faults across computational kernels while resulting in negligible overhead for fault-free executions.

Pragma-Based Compiler Extension for End-to-End Resiliency Against Soft Faults

by
Harsh Khetawat

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Master of Science

Computer Science

Raleigh, North Carolina

2017

APPROVED BY:

Dr. Hung-Wei Tseng

Dr. Guoliang Jin

Dr. Rainer Frank Mueller
Chair of Advisory Committee

DEDICATION

To my parents and sister.

BIOGRAPHY

The author was born in Calcutta, India. After completing his schooling in 2009 he started working towards his B.E. in Computer Science and Engineering at Manipal Institute of Technology. He graduated with an undergraduate degree in 2013, after which he worked for 2 years at Microsoft India Development Centre, Hyderabad in the Windows Networking team. In 2015, he began pursuing his graduate degree in Computer Science at North Carolina State University, Raleigh. Since January 2017 he has been working as a Graduate Research Assistant under the guidance of Dr. Frank Mueller.

ACKNOWLEDGEMENTS

I would like to thank everyone who has helped me over these last 2 years.

I would like to thank my advisor Dr. Frank Mueller for giving me the opportunity to work under his guidance. His deep insights, encouragement and advice have been invaluable towards the completion of this work. I would also like to thank Dr. Hung-Wei Tseng and Dr. Guoliang Jin for taking interest in my work and being part of my advisory committee. I would also like to thank the people in my research group with whom I've had truly wonderful discussions.

I would also like to thank Dr. Arash Rezaei for allowing me to work with him as part of his end-to-end resilience project. I would like to thank him for the design of the original resilience pragma and to identify the vulnerability of data structures during program execution. Without him this work would not have been possible.

I would like to thank my friends, Alok, Anas, Budhaditya, Dhruv, Mayank, Shubham and Ujan, who have made my life in Raleigh as enriching as it is and Akshit, Devanshi, Katrina and Monkey for their unceasing support and affection, often from very far away.

Finally, I would like to thank my parents for their unconditional love. Your strength and resolve are my greatest sources of inspiration.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER

1 INTRODUCTION

For large-scale parallel systems, faults have become the norm rather than the exception [SG06; Gei16; Bau05]. Bit flips and hardware faults often result in application or operating system failures. In order to make systems resilient against such failures, numerous hardware and software techniques have been devised [Mit05; San08]. It is projected that the frequency of faults will only exacerbate as we move towards future exascale systems. This might require a 20% increase in circuitry and energy in order to couter failures [Sni13]. With HPC systems moving increasingly towards commodity hardware, which are designed and built to be general-purpose, and not HPC-specific due to the manufacturing costs, a significant portion of the resiliency responsibility would likely fall on the software layer. Fig. 1.1 shows the number of soft errors encountered across DRAM modules from 3 different leading manufacturers as studied in Kim et al. [Kim14]. We can see that the number of soft errors have significantly increased since 2011 and might even be higher as we go towards higher density DRAM chips.

Prior research has emphasized the importance of resilience in future exascale HPC systems [Sni13; Cap09; Cap14]. What makes HPC systems particularly interesting to study are the multiplicity of challenges that they introduce. From size (millions of cores) to the tightly coupled programming model, combined with the scientific nature of applications which are highly parallel, often long-running jobs and have strict accuracy requirements makes HPC systems quite unique. With an increased number of components, the probability of failures also increases proportionally, and the tightly coupled programming model could result in quick fault propagation in one node to the other nodes in the system [Fia12a]. Furthermore, scientific applications require a high degree of accuracy and are often very sensitive to faults. These problems make resiliency a major roadblock on the path toward exascale HPC systems.

Memory hierarchies are becoming more complex with the introduction of persistent memory technologies such as Phase Change Memory (PCM), Magnetoresistive Random Access Memory (MRAM), and Memristors, and application programmers need to take into account the impact of these technologies on the resilience of their applications. This too creates new opportunities and challenges for developing resilience techniques that can enable next-generation scientific applications to fully exploit the capabilities of this novel hardware while still maintaining correct application execution.

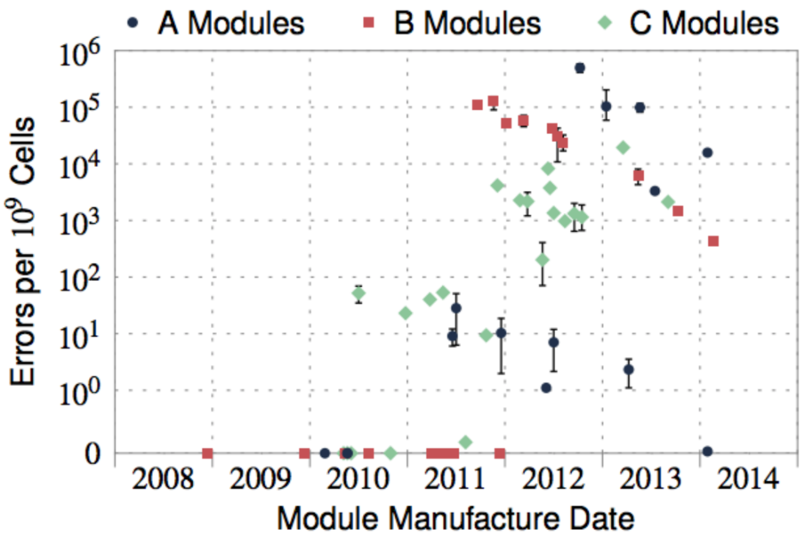


Figure 1.1 Normalized number of errors vs. manufacture date

1.1 Motivation: Combination of Software Resilience Techniques

Software resilience techniques are often used in practice in order to complement hardware protection against soft faults. To that end several software techniques such as checkpoint/restart (CR), redundant execution, and algorithm-based fault tolerance (ABFT) have been developed. Each of these techniques have their own limitations and benefits depending on the application. For example, CR needs to store checkpoints (storage overhead), and recover from a previous point in time when failures occur, which limits scalability [Ell12]. Similarly, redundant execution effectively doubles the amount of required resources [Fia12a]. On the other hand, ABFT has the benefit of low overheads but it needs to be customized for each numerical algorithm [HA84; Du12; Ell14a].

We can see that it is best to choose the resilience scheme on a per kernel basis rather than a single one for the entire application. This introduces a unique set of challenges. For one, it makes the job of the application programmer more difficult as the burden of combining these resilience techniques falls onto the application programmer. Secondly, it also creates windows of vulnerability between kernels protected by different resilience techniques.

We can consider two kernels, the first one with ABFT protection, and the succeeding one with redundant execution in which the second kernel consumes the result computed by the first kernel. Here, resilience is limited in scope to the boundaries of a kernel but does not extend across kernels to the entire program. This makes the results computed by the first kernel vulnerable till the second has consumed it. Furthermore, the composition of these two resilience techniques are non-trivial and results in additional effort for the application programmer.

1.2 Hypothesis

In order to tackle these challenges, we put forth the following hypothesis:

A pragma-based compiler extension for combining multiple resilience techniques can enable application developers to choose the resiliency technique on a per-kernel basis while also protecting the window of vulnerability between kernels for end-to-end resilience.

We propose to compose different low-cost techniques for data protection against soft faults across different application phases. End-to-end resilience creates protection spaces of kernels with different resilience techniques while eliminating the windows of vulnerability.

Resilience APIs, such as Containment Domains [Chu12], GVRGVR [Zhe14], and Charm++ [KK93], try to reduce the clutter created by meshing numerical methods with other resilience concerns, but cannot completely eliminate it. More general and transparent resilience methods, such as BLCR [Due03], have a much higher overhead than application-specific ones, such as CR or ABFT, whereas programs with application-specific resilience techniques are often hard to maintain.

A systematic resilience approach would be much more effective, especially for future systems, allowing application developers to focus entirely on algorithms and performance but delegate resilience requirements for seamless protection to the system (compiler/library/runtime). A pragma-based scheme can therefore be used to provide end-to-end resilience using the benefits of aspect-oriented programming (AOP) [Kic97]. It results in increased modularity by separating the algorithmic and resilience concerns while supporting a variety of paradigms and methods.

We implement an OpenMP pragma extension to support the separation of the algorithmic and resilience aspects in the Cetus compiler [Dav09], develop a fault injection mechanism that can generate faults at a given average rate, and finally evaluate applications for end-to-end protection against the injected faults. Our results show that end-to-end protection has negligible overhead for fault-free execution, and manages to detect and correct affected data structures even between kernels.

CHAPTER 2

BACKGROUND

2.1 Failures

Avizienis et al. [Avi04] define faults as the cause of errors, which, if not detected or corrected, may lead to failures. A failure could manifest itself as an incorrect result (which may go undetected), a crash, or an interruption in execution. With an increase in the rate of failures, forward execution of the application is hindered, which impedes scientific progress in HPC systems. This has considerable cost impacts both in terms of time and energy. Failures can be distinguished into hardware and software failures

2.1.1 Hardware Failures

Hardware faults can be either persistent or transient. Persistent faults are those faults that are generally caused by aging hardware, or operation of the hardware beyond the manufacturers thresholds. They are usually resolved by replacing the concerned hardware device. Failures caused by these faults result in the application being interrupted and might result in the executing job being rendered useless.

Transient faults, in contrast, often occur due to cosmic radiation. The interaction of high energy neutrons with the silicon die on the device causes a cascade of charged particles. The pulse created by these particles might be powerful enough to flip bits in memory. The decrease in size of transistors and the reduced charge they carry increases the probability of multiple bit flips [Ash10]. These faults might not necessarily stop application progress but could very well result in invalid data and results. Several data paths (caches, data bus, etc.) are protected by ECC, which can correct single bit flips but not multiple bit flips. For perspective, Jaguar's 360TB of DRAM experienced a double bit flip every 24 hours [Gei16]. Jaguar uses chipkill which can correct some double bit flips.

Multiple bit flips might cause some faults to go undetected and result in Silent Data Corruption (SDC). They manifest as incorrect interim or final results for the application. Studies have shown that SDC rates are often orders of magnitude larger than those specified by the manufacturers [PS07; Sch09; Sri15]. Another study by Schroeder et al. [Sch09] has concluded that over 8% of DIMMs are affected by errors over the course of 2.5 years on large-scale platforms. Furthermore, Sridharan et al. [Sri15] have analyzed the effectiveness of techniques such as ECC an chipkill. Another study by Microsoft [Nig11] of over a million consumer PCs also shows that CPU faults are quite frequent. A report from Argonne National Laboratory (ANL) [Sni13] stresses the need to do further research on SDC rates and calls for refinements in hardware and software techniques to handle them.

2.1.2 Software Failures

Software failures are due to bugs in software (code), many of which only materialize when executing at scale [Cot10]. Interactions between complex distributed components and race conditions between parallel tasks are often cited as reasons for software failures. With applications being developed for higher performance, new complexity is introduced, making the codes more error-prone [Ash10]. The report from ANL [Sni13] has also predicted that an exascale platform could potentially experience a failure every 30 minutes.

2.2 Resilience Methods

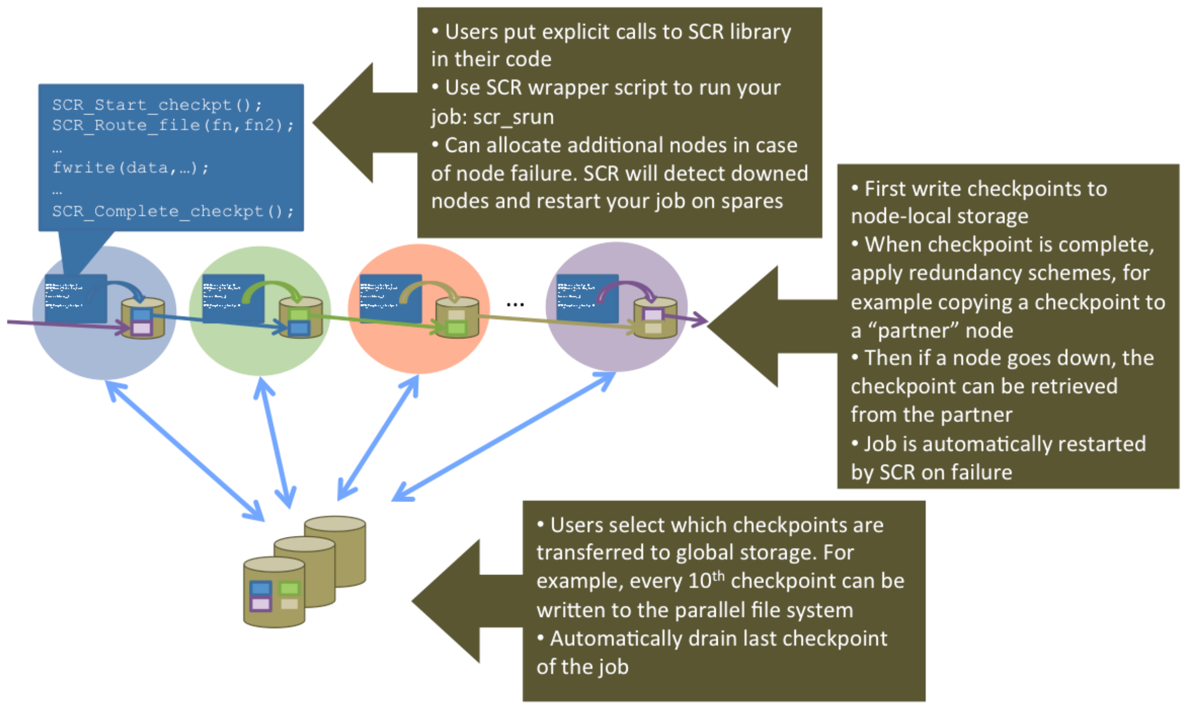
Due to the prevalance of faults in HPC systems, resilience has become a necessity for protecting applications. These methods are used to compensate for state loss due to failures by performing a forward or backard recovery. Backward recovery recreates an older state of the application by rolling back using either system-level or application-level checkpoint/restart (CR) mechanisms [Moo10a]. Forward recovery, in contrast, repairs the affected data structures. This can be done by either recovering the values from a replica (redundant computing) [Fia12a], or using Algorithm-Based Fault Tolerance (ABFT) from checksums or other algorithm-specific properties [HA84; Du12; CW14; Sha12; Ell14a].

2.2.1 Algorithm-Based Fault Tolerance (ABFT)

ABFT methods utilize a deep understanding of the algorithm and use an inherent property specific to the algorithm or data structure in order to recover from errors. ABFT for algorithms, such as matrix multiplication [HA84; CD08], Cholesky decomposition [CW14], sparse linear solver [Sha12], and matrix factorization [Du12], have been proposed. Studies have shown merits of these techniques based on the data or algorithm, implying that future systems could use a mix of these techniques on a per-phase basis throughout the execution of the application.

2.2.2 Checkpoint/Restart (CR)

In checkpoint/restart (CR), applications are periodically checkpointed, i.e., the state of the application is written to persistent storage. When a failure occurs the application is rolled-back to the previous state for all the tasks and execution resumes from that point. Several improvements for CR have been proposed in order to tackle the overhead. Techniques such as data aggregation [Isl12] and data deduplication [Bis11] reduce the size of the checkpoint. In contrast, multi-level checkpointing schemes [Moo10a; BG11] and in-memory checkpointing [Zhe12] improve the speed and scalability of checkpoint/restart. Fig. 2.1 gives an overview of the Scalable Checkpoint/Recovery (SCR) [Moo10a] library developed by Lawrence Livermore National Laboratory (LLNL). Amongst other capabilities, it features multi-level checkpointing, has support for protecting checkpoints from device failures, and also compression of checkpoints for efficiency.



```
SCR_Start_checkpt();
SCR_Route_file(fn,fn2);
…
fwrite(data,…);
…
SCR_Complete_checkpt();
```

- Users put explicit calls to SCR library in their code
- Use SCR wrapper script to run your job: scr_srun
- Can allocate additional nodes in case of node failure. SCR will detect downed nodes and restart your job on spares

- First write checkpoints to node-local storage
- When checkpoint is complete, apply redundancy schemes, for example copying a checkpoint to a “partner” node
- Then if a node goes down, the checkpoint can be retrieved from the partner
- Job is automatically restarted by SCR on failure

- Users select which checkpoints are transferred to global storage. For example, every 10th checkpoint can be written to the parallel file system
- Automatically drain last checkpoint of the job

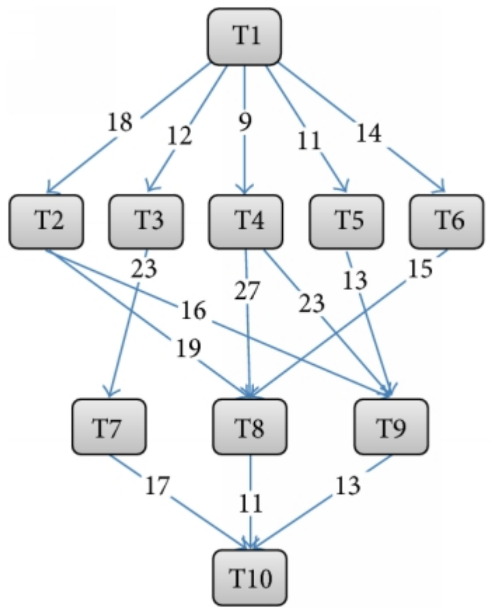
Figure 2.1 Overview of the Scalable Checkpoint/Restart library

2.2.3 Redundancy

Yet another commonly used resilience technique is redundant execution [Bri10; Fer11; EB11; Fia12a]. Here, multiple instances are utilized to perform the same task. The results are then compared to ensure failure free execution. When double redundancy is used, the results are compared to make sure they match, whereas triple redundancy (with voting) can be used to correct them. Marathe et al. [Mar14] have demonstrated that redundancy can be a viable approach for HPC systems in the cloud.

2.3 End-to-End Resilience

Scientific applications are often structured as a multi-phase pipeline with each phase representing a kernel. The previously described methods are designed to work well with certain kernels but have significant overhead for others. A one-size-fits-all approach might often lead to significant overheads, which can potentially be avoided with a per phase resilience technique. The method that works best can be chosen on a per-kernel basis keeping in mind factors such as data size and computation time. As part of this work, we propose a compiler technique that enables end-to-end resilience for such applications while allowing the programmer to choose a preferable resilience method for each kernel. End-to-end data integrity has also been cited as goal in the exascale report by ANL [Sni13].



Task	P_1	P_2	P_3
T1	14	16	9
T2	13	19	18
T3	11	13	19
T4	13	8	7
T5	12	13	10
T6	13	16	9
T7	7	15	11
T8	5	11	14
T9	18	12	20
T10	21	7	16

Figure 2.2 Example of a Task-based Schedule Represented as a Directed Acyclic Graph

2.4 Task-based Scheduling

Task-based scheduling for multi-cores has been extensively studied previously [Che07; AC06]. The basic premise is to divide the application into a set of tasks that are executed by resources in a certain order depending upon the dependency constraints. Fig. 2.2 shows an example of a schedule along with the cost matrix. Metrics such as throughput [Mar03; Leo05], fairness [And04; Lee09], response time [Che09; Che12], etc. have been studied with respect to task-based scheduling. These metrics are primarily performance-driven.

Such a paradigm is often represented as a Directed Acyclic Graph (DAG) comprising of tasks and their dependencies. During execution, this DAG can be traversed in either a breadth-first or depth-first manner or both. Each of these has a specific impact on the parallelism, cache locality, and the vulnerability of the data structures. As part of this work we integrate end-to-end resilience with both OpenMP-based parallel execution and a task-based runtime and finally contrast them.

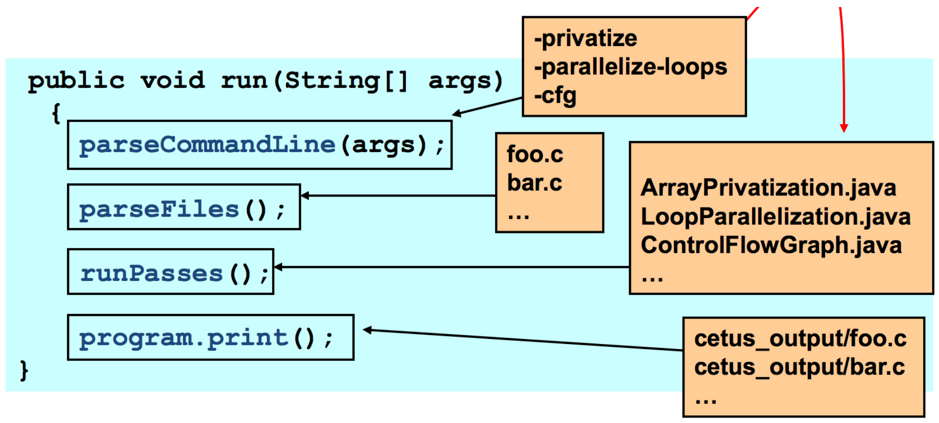


Figure 2.3 The Cetus Driver

2.5 Source-to-source Compilation

A source-to-source compiler transforms the source code of a given input program into a modified source code. We have designed a compiler directive as an extension to the OpenMP framework that allow the application programmer to specify the resilience characteristics of each kernel. We then use source-to-source compilation to transform the source code into an application with end-to-end resilience capabilities.

After investigating alternatives like GCC and Insieme, we decided to implement the directive as a transform pass on Cetus. Developed by researchers at Purdue University, Cetus [Dav09] is a compiler infrastructure for the source-to-source transformation of programs, particularly ANSI C in our case. The Cetus project is 80,000 lines of Java code. Fig. 2.3 shows the basic execution of the Cetus compiler. Cetus uses Antlr in order to parse C programs into Intermediate Representation (IR). The runPasses() function then serially runs the passes on the IR in order to generate the output source code. Each pass iterates over the IR and is capable of modifying it by adding, removing, or editing the input source. New code is added as Cetus IR objects, which is equivalent to building an IR tree from its leaves. Similarly, complex IR can be generated by extending these trees. Cetus allows iterating through the IR in both, a depth-first and a breadth-first manner. In our implementation, we use depth-first iteration in order to traverse the input IR.

CHAPTER

3 RELATED WORK

A number of hardware and software techniques have been proposed in order to tackle the problem of SDCs in HPC systems. Mitra et al. [Mit07] have proposed a hardware technique, Built-in soft error resilience (BISER), which is an architecture-aware circuit design technique that can correct soft errors in computational circuits. It reduces the chip-level soft error rate by an order of magnitude with minor overheads. Similarly, Kelin et al. [Kel10] have proposed a chip layout design principle, Layout Design through Error-Aware Transistor Positioning (LEAP), which goes beyond traditional layout techniques and makes the digital circuits significantly more resilient against soft faults. Other solutions, such as Cross-layer Exploration for Architecting Resilience (CLEAR) [Che16], have also been proposed. CLEAR combines resilience across various layers of the system, such as circuit, logic, architecture, and software, etc., in order to achieve the desired resilience targets.

Cao et al. [Cao15] have proposed a software technique to make dynamic task-based runtimes resilient to soft errors. On detection of an error, it recovers the application by re-executing the minimum required sub-DAG. It also keeps checkpoints of data between tasks in order to minimize the re-execution in case of failure, and also leverages insights about the algorithm in order to recover the data without re-execution. Furthermore, Elliot et al. [Ell14b] argue that numerical methods can benefit from a numerical unreliability fault model. This would allow the application to use cheap sanity checks in order to bound errors in computational results.

Fiala et al. [Fia12b] have proposed a transparent MPI redundancy scheme that would allow faults to be detected and corrected transparently. It detects faults by comparing the MPI messages between a pair of replicas. If the replicas have differing messages, corrective action can be taken. They also show that a single error has the potential to cause a cascading pattern of corruption that can spread to other processes. FlipSphere [Fia16] is another software-based DRAM error detection and correction library that uses hardware accelerators to increase application resilience. It implements on-demand verification of page integrity, and recovery based on an error correcting code in case of faults.

Another software-based technique proposed by Chung et al. [Chu12] uses containment domains as a programming construct to express the resilience needs of parts of the application, along with error detection, state preservation, and recovery mechanisms. It allows the programmer to exploit the hierarchical nature of the application by hierarchical state preservation, restoration, and recovery. Other resilience methods, such as checkpoint/restart [Moo10b; Fer12; Sat12; Joh90], and redundant computing [Bri10; EB11; DÏ2], can also be used to protect against SDCs.

CHAPTER

4 DESIGN

HPC applications often consist of phases of computation. Each phase takes certain inputs, consumes them, and generates the output from that phase. These outputs are themselves intermediate results that are consumed as input by the next phase of the application. Using our pragma approach we aim to exploit the live range of these variables in and across each of the phases. On the last use of these variables we can perform a check for correctness that enable us to guarantee the integrity of this variable from when it was defined up to its last use.

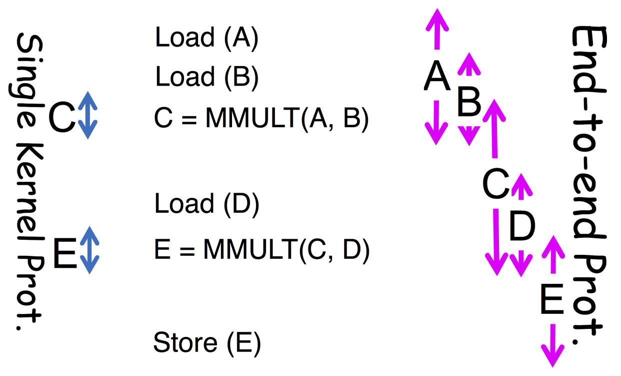


Figure 4.1 Protection of variables with conventional resilience vs end-to-end resilience

Fig. 4.1 shows a sequentially composed matrix multiplication application. Huang et al. [HA84] protect each of the matrices by adding another row and column to the matrix that is used to store the calculated row sums and column sums, respectively. In this case protection is provided for the result matrix in each matrix multiplication kernel. With end-to-end resilience we can provide protection to the matrices throughout their live-ranges. While single kernel approaches are constrained to the kernel boundaries, the end-to-end approach does not have this limitation and can span across kernel boundaries.

4.1 The Protect Pragma

In order for the application developer to specify the resilience characteristics of each kernel in a simplified manner we propose the following pragma scheme. The source-to-source compiler can use pragma directives to expand the code to provide end-to-end resilience. The expanded source code performs checks on the data structures and takes corrective action if necessary. In order to facilitate adoption and code maintainance, and for possible future synergy between thread parallelism and resilience, we incorporate the protect pragma directive into the OpenMP library.

4.1.1 Anatomy of a Protect Pragma

The following is an example of a typical protect pragma:

$$\#pragma\ omp\ protect\ [M]\,[Check(f_1,...,f_n)]$$
$$[Recover(g_1,...,g_m)]\,[Continue]$$

[M] is an optional argument that represents the resilience method to use. "Redundancy" is used in case one or more shadows are being used as part of the runtime. A shadow refers to a redundant MPI rank. "CR" is used to indicate that a checkpoint needs to be created at that point. With the "Check" clause one needs to specify a comma separated list of the checker functions that is called for checking each data structure that requires protection. Similarly, the "Recover" function is a comma separated list of functions to recover the data structures in case the check fails. The Continue keyword is used to indicate to the compiler that the data structure is live beyond the current region. This informs the compiler that the data structure requires end-to-end protection.

The pragma directive is used to expand the source code it annonates into a while loop that enables consistency checks and recovery operations. The flow of control in the program is allowed to exit the while loop only when internal data consistency has been assured. In case recovery is not possible, the program exits.

Some data structures in the program might depend on other data structures. They can be recovered by recomputation, which is only feasible if none of the constituent data structures had been corrupted, or recomputations would produce incorrect results. Analysis of the definition and use of these structures allows us to effectively chain these directives to other directives. Such a chaining, once pragma-expanded, results in nested while loops to establish end-to-end protection.

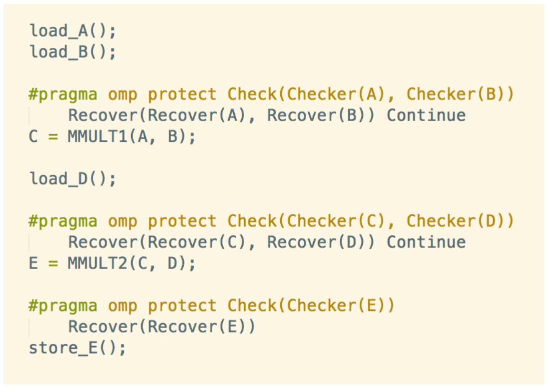
4.2 Task-based Resilience

Similar to resilience in OpenMP-based parallel kernels, we also present a task-based resilience scheme. As tasking libraries are becoming more popular in the HPC community, there is a growing need for efficient ways to provide resilience to task-based applications. For such applications one can check the correctness of each fine-grained task. In case we detect an error in the result of this task, data correction is attempted. In case correction is not possible data is recomputed, but only for the required subset of tasks to fix this data. This provides fine-grained and efficient resilience. It also allows application developers to specify resilience techiniques on a per-task basis.

4.3 Source-to-Source Compilation using Cetus

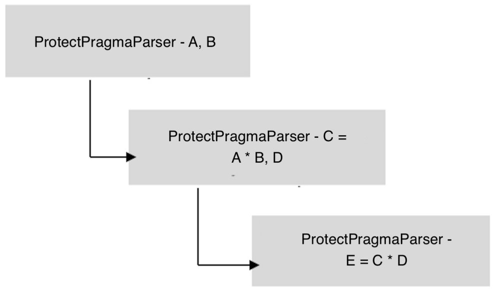
We designed and implemented support for out pragma expansion in Cetus [Dav09]. We added to Cetus the ProtectPragmaParser class, a transform pass that implements our pragma. Each pragma directive in the input program is represented as an object of the ProtectPragmaParser class.

The ProtectPragmaParser class is used to transform the generated parse tree to an equivalent parse tree structure that contains our protection boundaries, checker functionality, and recovery mechanisms. We traverse the input parse tree in a depth-first manner looking for the protect pragma directives. On finding the pragma, we parse the directive to populate the checker and recovery functions associated with this particular pragma. We also generate the necessary protection boundary, checking, and recovery code required in the current context and track the variables defined at these protection boundaries.



```
load_A();
load_B();

#pragma omp protect Check(Checker(A), Checker(B))
    Recover(Recover(A), Recover(B)) Continue
C = MMULT1(A, B);

load_D();

#pragma omp protect Check(Checker(C), Checker(D))
    Recover(Recover(C), Recover(D)) Continue
E = MMULT2(C, D);

#pragma omp protect Check(Checker(E))
    Recover(Recover(E))
store_E();
```

(a) Code for sequentially composed matrix multiplication along with the pragma


ProtectPragmaParser - A, B
ProtectPragmaParser - C = A * B, D
ProtectPragmaParser - E = C * D

(b) Logical structure of pragma instances created from the input program

Figure 4.2 Input code and the generated logical structure

As part of the ProtectPragmaParser object creation, we check if the current directive is chained to a previously encountered directive via the `Continue` keyword. If chained, we can recompute these variables in case their recovery methods fail, and the ProtectPragmaParser object of the current context is added to the list of chained pragmas of the directive it is chained to. Otherwise, it is added as an independent (root) pragma. When chaining is found in the input IR, we extend the protection boundaries of the current pragma around that of the following pragma. When the input source code has been completely parsed, a logical structure of these chained (or unchained) directives is created in memory. An example of this logical chained structure is shown in Fig. 4.2b for code in Fig. 4.2a.

The logical structure has one root pragma object and two chained pragma objects as is evident from the input source code. Finally, the generated lines of code are added to the IR, and the final source code is produced as output. Fig. 4.3 shows an example of how new IR is created in Cetus.

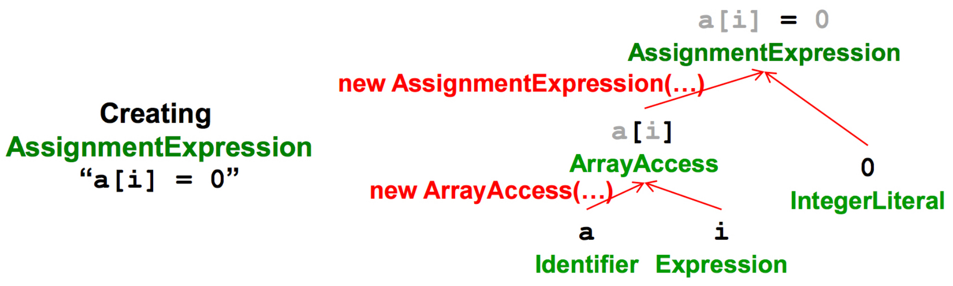


Figure 4.3 Creation of new IR elements in Cetus

The Cetus compiler infrastructure, along with our ProtectPragmaParser functionality, allows us to transform our input source code in this manner to support end-to-end resilience. While these transformations could be performed manually by the programmer for simple examples, it quickly becomes tedious and error-prone for more complicated program structures or even chained regions. Our Cetus implementation transforms the input source in a single pass through the IR tree, emitting code recursively even for complicated, inter-leaving dependencies between resilient variables. This supports the development of powerful software that has end-to-end resilience while off-loading the repetitive and generally non-trivial task of code expansion to the compiler.

4.4 Fault Injection

We simulate the effects of faults on our test applications by implementing a fault injection framework. The fault injection framework is capable of injecting faults at random intervals while maintaining an average Mean Time Between Faults (MTBF). The average MTBF can be specified in seconds.

We implement the fault injector as a function that is executed periodically at a random interval (0, 2 * MTBF), where MTBF is the mean time between faults specified before the start of the execution. In order to execute this function at these intervals, we implement a signal handler, called injectHandler, which is called in the given interval. In turn, the injectHandler also schedules another signal in that interval after it injected the fault.

Every time the injectHandler is called, it randomly picks a data structure to inject a fault into. For matrix multiplication the probability of a fault occuring in any of the data structures is the same as they are the same size. Similarly, for TF-IDF we inject faults uniformly over the data structures and present the normalized number of faults based on their respective data structure sizes. Furthermore, after selecting the data structure, the byte subject to injection is also selected randomly.

The random nature of the injector in terms of frequency, choice of data structure, and the point of injection allows our end-to-end resilience technique to encounter a variety of possible real-world faults. This also allows us to study injections that happen to data live in the kernel, live data across kernels, and stale data.

CHAPTER

5 IMPLEMENTATION

In this section, we describe the implementation of the transform pass that expands the pragma-annotated code to support end-to-end resilience. The pragma expansion is done in a transform pass in the Cetus compiler, ProtectPragmaParser, which takes as input the IR of the program and converts into an equivalent IR with the protection mechanisms. Cetus can then ouput the program that is represented by the transformed IR. We will also describe the implementation of the fault injector we use in order to introduce faults in the data structures at a given rate.

5.1 ProtectPragmaParser Class

The ProtectPragmaParser class is a transform pass that implements our pragma. Each pragma directive in the input program is represented as an object of the ProtectPragmaParser class. The compiler parses the input IR looking for the `protect` pragma and creates a logical structure of these chained pragmas in memory. The chained pragmas are expanded at protection boundaries to trigger the checking and recovery that realizes end-to-end protection of the data structures.

Once the entire input source code has been traversed and the logical structure of pragmas is created, a recursive function that emits transformed code is invoked on the root objects. This, in turn, invokes the function on each of its chained pragmas. It is at this stage that checking and recovery code for non-last-use variables is removed so as to reduce the checking overhead. The `emitCode` uses the chaining information to correctly emit the nested while loop structure in the expanded code. As part of the code emitting process, if a particular directive had the `CR` or `Redundancy` clause, then the compiler emits the appropriate function calls to `wake_shadow` and `sleep_shadow` in case of the `Redundancy` clause, and `create_ckpt` in case of the `CR` clause.

We now present the properties of the ProtectPragmaParser class and the algorithms that are responsible for parsing and transforming the input source code to an equivalent code that realizes end-to-end resilience.

5.1.1 Member Variables

5.1.1.1 rootPeerPragmas

This is a static member of the ProtectPragmaParser class. It is a list of all the independent pragmas that are not chained to a previous pragma. The `emitCode` function is explicitly called for each of the objects contained in this list. From Fig. 5.1, the first pragma would be added to the list of `rootPeerPragmas` as it is not chained to a previous pragma.

5.1.1.2 chainedPragmas

This member contains a list of all the pragmas that are chained to the current one. Each root pragma contains the chainedPragmas object. The `emitCode` function is called recursively for these pragmas from the call to the `emitCode` function from the root pragma. In Fig. 5.1 the second pragma would be added to the `chainedPragmas` list of the first pragma, and the third pragma would be added to the `chainedPragmas` list of the second.

5.1.1.3 annotObj

This is a Traversable object that contains a reference to the IR object that the current pragma annotates. We store the annotated object as part of the pragma in order to determine the variables that are used and defined as part of the current pragma. For each of the pragmas from Fig. 5.1, the `annotObj` will be the function call to the kernel annotated by the pragma.

5.1.1.4 checkers

The `checkers` object contains a map of variables to its respective checker functions. The annotated statement and the pragma directive are parsed to determine the checker functions that would be required in order to check the respective data structure for integrity. The `Check` functions in each pragma from Fig. 5.1 will be added to the the `checkers` map of the respective pragmas.

5.1.1.5 recoverers

The `recoverers` object is a map of variables to a list of its respective recovery functions. Similar to the checkers, the annotated statement and the pragma directive are parsed to determine the list of recovery functions to be used to recover the data structure in case the integrity check fails. The `Recover` functions in the pragmas from Fig. 5.1 are added to the `recoverers` maps of their respective pragma objects.

5.1.1.6 redundFlag and CRFlag

`redundFlag` and `CRFlag` are boolean variables we use to track whether the current pragma has the optional redundacy or CRFlag clause. The `emitCode` function checks these boolean variables to determine if we need to create IR objects for redundacy and checkpoint/recovery.

5.1.1.7 protectLoop

`protectLoop` is an IR object of type WhileLoop that represents the entire protection boundary of this pragma directive. The boolean check array, the kernel function call, the checking and recovery calls for each data structure, and the final check to ensure the integrity of all data structures are added to this while loop block.

```
#pragma omp protect Check(Checker(A))  Recover(Recover(A)) Continue
B = KERNEL1(A);

#pragma omp protect Check(Checker(A), Checker(B))  Recover(Recover(A)) Continue
C = KERNEL2(A, B);

#pragma omp protect Check(Checker(C)) Recover(Recover(C))
KERNEL3(C);
```

Figure 5.1 An example multi-kernel program with pragmas for end-to-end resilience

5.1.2 Member Functions

5.1.2.1 start

This static method is the entry point into our pass. It iterates over the IR and scans for our pragma. When the transform pass encounters a protect pragma, it creates a ProtectPragmaParser object for the directive. The constructor is responsible for parsing the text of the annotation in order to derive information about the "Check" and "Recover" functions, including other flags. It then populates the map of data structures to their respective checker and correction functions. Depending on the definition and use of these data structures, this method checks if this annotation can be chained to a previously encountered directive. Once the input source code has been completely traversed, we have a logical structure of these chained (or unchained) directives in memory. This is shown in Algorithm 1, where lines 7-8 scan and parse the input pragma directive, line 10 calls the constructor in order to create the ProtectPragmaParser object, and lines 12-19 check if the current pragma needs to be chained to a previous pragma directive.

Algorithm 1: Start of the compiler pass

```
Function start
    rootPeerPragmas ← new ProtectPragmaParser list
    iter ← get IR iterator
    while iter has elements do
        obj ← IR object currently at iter
        if obj is of type Annotation then
            tokenArray ← split pragma string
            Parse tokenArray to determine characteristics of pragma
            if pragma is of type protect then
                Create ProtectPragmaParser object by parsing checkers, recoverers, and
                  clauses
                Check if there is an existing context from a previous pragma
                if previous context exists then
                    Check if chaining is required
                end
                if chained then
                    Add this ProtectPragmaParser object to chainedPragmas list of the other
                      ProtectPragmaParser object
                end
                else
                    Add this ProtectPragmaParser object to the static rootPeerPragmas list
            end
        end
    end
    Add completed boolean array and initialize to false
    foreach ProtectPragmaParser object in rootPeerPragmas do
        Call emitCode on each object
    end
```

5.1.2.2 findChaining

This static method takes two ProtectPragmaParser objects as parameters and determines if they can be chained to each other. If chaining is possible, then the pragma is added in the chainedPragmas list of the other pragma, otherwise it is added in the static rootPeerPragmas list. To check chaining, this function determines if the variables that are used in this pragma's annotated statement have been defined as part of any other pragma's statement from the current context.

5.1.2.3 chain

This method, pseudocode shown in Algorithm 2 chains the current pragma object to the specified pragma object. This is done in case the findChaining function determines that the specified pragma can be chained to the current one. It expands code for the initialization, looping, checking, and recovery code generated for the current pragma within the protection loop of the specified pragma object. It also emits the necessary function calls for redundancy and checkpoint/recovery mechanisms in case those clauses are specified in the pragma.

Algorithm 2: Chaining for end-to-end protection

```
1  Function chain
2      Create a new protection block
3      Add the check array declaration to the block
4      Add the protect loop to be chained to the block
5      if CRFlag is true then
6          Add a function call to create_ckpt to the block
7      end
8      if redundancy flag is true then
9          Add the function calls to wake_shadow and sleep_shadow to the block
10     end
11     Add check to confirm correct execution of this block
12     Create a while loop object with this protection block and all check conditions are satisfied
```

5.1.2.4 validateLastUse

This recursive method is used to ensure that checking and recovery operations for a certain variable is done only on its last use. It recursively removes map entries for variables from its chained pragmas that the current pragma has in its checker map.

5.1.2.5 getLastDef

This recursive function returns the ProtectPragmaParser object that annotates the definition for a variable defined by the current pragma. This ensures that recovery for a variable defined by this pragma is chained to a previous pragma that also defines the same variable. In case such a pragma cannot be found, recovery cannot be done, and we add code in the IR to clean up and exit.

5.1.2.6 emitCode

Once the input source code has been completely parsed, this structure of chained directives is used to emit code through a recursive function. The function uses the chaining information to emit the nested while loop structure as part of the code expansion. Furthermore, the chain function is called, which emits the appropriate function calls to `wake shadow` and `sleep shadow` in case of the `Redundacy` clause, and `create ckpt` in case of the `CR` clause.

The use of a recursive function to emit code allows us to generate a complex structure of nested while loops while only iterating for a single pass through the input source code. Algorithm 3 shows the pseudocode for the `emitCode` function. Here, lines 5-20 add the checking and recovery functionality to the output IR and create the protection boundaries for the current pragma, lines 21-26 are responsible for adding the function calls for checkpoint/recovery and redundancy in case those clauses are specified, and lines 31-33 call the `emitCode` function on the chained pragma directives rescursively.

Algorithm 3: Output code with end-to-end protection

```
Function emitCode
    Call validateLastUse to remove checking for variables that are live after the current use
    Create boolean check array of size equal to number of checker functions in pragma
    Create protection block for the current pragma
    foreach variable in checkers map do
        Create a function call to the checker function
        Create assignment statement to save the return value to the check array
        Add statement to the protection block
        foreach recover function for current variable do
            Create a function call to the recovery function
            Create IR object to check return value of recovery function
        end
        Create IR object for recovery block
        Add recovery function calls to the recovery block
        Add code to assign false to the completed array element in case recovery has happened
        Add code to clean-up and exit if recovery is not possible
        Create IR object to add check condition at the end of the protection block
    end
    Add IR to check if all checking functions return true
    Add IR to assign true to completed array element if all checking functions return true
    if CRFlag is true then
        Add a function call to create_ckpt to the block
    end
    if redundancy flag is true then
        Add the function calls to wake_shadown and sleep_shadow to the block
    end
    if chaining is required then
        Call the chain function to chain protection boundaries
    end
    Assign while loop with the protection block to protectLoop
    foreach ProtectPragmaParser object in the chainedPragmas list do
        Call emitCode on the ProtectPragmaParser recursively
    end
```

5.2 Fault Injection

We implement a fault injector in order to verify the end-to-end resilience capabilities of applications enhanced with our pragma directives. We implement a signal handler to handle `SIGALRM` signals using the POSIX compliant `sigaction` system call. The `sigaction` function is called in the initialization phase of the application. We then call the `alarm` function and randomly choose a time within the interval of [0, 2 * MTBF] as the time after which the first `SIGALRM` signal is issued.

The signal handler is responsible for randomly selecting a data structure. After a data structure is selected, the signal handler calls the `inject` function. It is an application specific function to inject faults into the selected data structure. The signal handler also schedules another `SIGALRM` signal to be issued at a random time within the interval of [0, 2 * MTBF]. This causes regular `SIGALRM` signals to be issued until the end of the application execution. Algorithm 4 shows the steps of signal handling. In this algorithm, line 1 selects the random data structure to inject the fault into, line 2 keeps a count of the number of faults injected into each data structure, line 3 calls an application-specific function to inject the fault, and lines 5-6 select and schedule the next fault injection.

Algorithm 4: Fault Injection

```
1 Function injectionHandler
2     $ds \leftarrow$ random data structure
3     $injCount[ds] \leftarrow injCount[ds] + 1$
4     Inject fault in ds
5     $nextSignal \leftarrow$ random value within [0, 2 * MTBF]
6     Schedule another signal in nextSignal seconds
```

CHAPTER

6 EXPERIMENTAL EVALUATION

We evaluate the end-to-end resilience using matrix multiplication, and TF-IDF. The matrix multiplication application consists of two successive matrix multiplication kernels as previously shown. We also evaluate task-based matrix multiplication, where we refactor the kernels for fine-grained tasking. We evaluate the effectiveness of end-to-end resilience on TF-IDF using the pragma based approach to combine redundancy and checkpoint/recovery. Finally, we present the performance under different fault injection scenarios, fault-free overhead, and compare the performance between conventional and end-to-end resilience.

6.1 Experimental Setup

We conducted our experiments on the ARC cluster. The cluster has 108 nodes, each with two AMD Opteron 6128 processors (16 cores total) and 32GB of RAM running CentOS 7.3 and Linux 4.10. The cores are running at 2.0GHz, and have 512KB of L2 cache and 128KB of L1 data/instruction cache per core. For the pragma-based sequentially composed matrix multiplication, we use OpenMP in order to parallelize the matrix multiplication kernels, and ABFT on a per-kernel basis as the resilience technique. The task-based matrix multiplication is parallelized using POSIX threads with ABFT for resilience on a per-block basis. The TF- IDF application with its two kernels, TF and IDF, uses

OpenMPI to parallelize across nodes with both checkpoint/recovery for one kernel and redundancy for the other kernel for resilience.

6.2 Matrix Multiplication

We evaluate two variants of the matrix multiplication application, pragma-based and task-based. For each of the variants we choose 5 input sizes varying from 512x512 to 2560x2560. The OpenMP parallelized pragma version uses 16 OpenMP threads that perform matrix multiplication in a blocked manner with a block size of 32x32, whereas the task-based version schedules smaller tasks using a dependency graph.

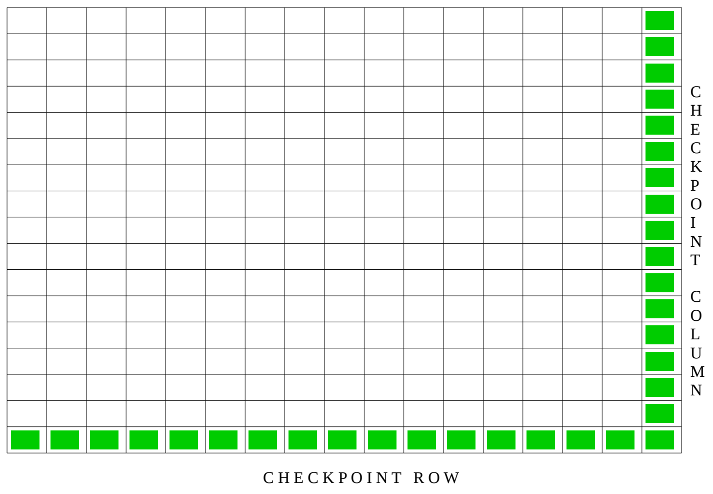


Figure 6.1 Checksum row/column in ABFT protected matrix

6.2.1 Resilience Technique

Both variants of matrix multiplication uses ABFT to provide resilience to their kernels. Fig. 6.1 shows the extra row and column added to a 16x16 matrix for the pragma-based variant in order to store the checksum for checking and recovery. Each element in the checksum row/column contains the sum over all the elements in that row/column. When a fault occurs in the matrix, the sum of the row and the column of that element does not match with the stored sum. This allows the check method to detect the location of the fault in the matrix. Furthermore, faults can be corrected using the difference in the calculated sum and the stored sums as long as only one element per row/column pair is compromised. Otherwise, an entire result matrix needs to be recalculated from its inputs.

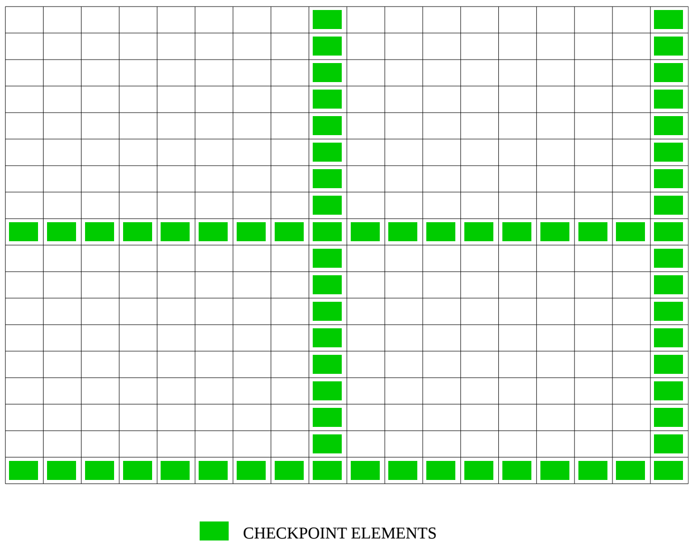


Figure 6.2 Checksum row/column in ABFT protected matrix

For task-based matrix multiplication, we store checksums for each block. This allows for fine-grained checking and recovery. The tasks are responsible for calculating a certain block. Single bit flips can be corrected within a row/column pair of a block. Upon multiple flips per row/column, a block is re-calculated, not the entire matrix. Fig. 6.2 shows an example of a 16x16 matrix with 8x8 blocks.

6.2.2 Pragma Expansion

To provide end-to-end resilience to the pragma-based variant of the sequentially composed matrix multiplication, we utilize our source-to-source compilation pass on Cetus. For the code shown in Fig. 4.2a, the transformed code is shown in Fig. 6.3.

```
load_A();
load_B();
bool completed[3]={false}, first=true;
while(!completed[2]){
  while(!completed[1]){
    while(!completed[0]){
      bool check[2];      //2 check flags
      do{
        C = MMULT1(A,B);  //OpenMP threads
        if (!(check[0] = Checker(A)))
          if (!Recover(A)){  //Correct(A)/Load(A)
            throw unrecoverable;
          }
        if (!(check[1] = Checker(B)))
          if (!Recover(B)){  //Correct(B)/Load(B)
            throw unrecoverable;
          }
      }while(!check[0] || !check[1]);
      if (check[0] && check[1])
        completed[0]=true;
    }
    if (first) {
      load_D();
      first = false; // load D exactly once
    }
    bool check[2];     // 2 check flags
    do{
      E = MMULT2(C,D); //OpenMP threads
      if (!(check[0] = Checker(C)))
        if (!Recover(C)){  //Correct(C)
          completed[0]=false;
          break;
        }
      if (!(check[1] = Checker(D)))
        if (!Recover(D)){  //Correct(D)/Load(D)
          throw unrecoverable;
        }
    }while(!check[0] || !check[1]);
    if (check[0] && check[1])
    completed[1]=true;
  }
  bool check[1]; // 1 check flag
  do{
    store_E();
    if (!(check[0] = Check(E)))
      if (!(Recover(E))){ //Correct(E)
        completed[1]=false;
        break;
      }
  }while(!check[0]);
  if (check[0])
   completed[2] = true;
}
```

Figure 6.3 Code generated from pragmas for sequentially composed matrix multiplication

6.2.3 Fault Injection

For both the variants of the matrix multiplication application, we use the previously described fault injector in order to study the performance of the application under faults. We inject faults by first randomly selecting a matrix. Since all the matrices are of the same size, the probability of selecting any one of them is the same. Then we select a random element from that matrix for injection. In order to finally inject a fault, we XOR the selected element with a random value of a single bit that is set. The fault injector is executed at random time intervals while maintaining a given mean time. This allows more than one fault to be injected in a single execution of the application.

6.2.4 Performance Evaluation

We evaluate the performance of our end-to-end resilient matrix multiplication over a 100 executions of the application. We select five matrix sizes, 512x512, 1024x1024, 1536x1536, 2048x2048 and 2560x2560, and three time intervals, 25, 35 and 45 seconds, for the mean time between failures (MTBF). While such high fault rates may be unlikely, they help us to assess the robustness of our technique.

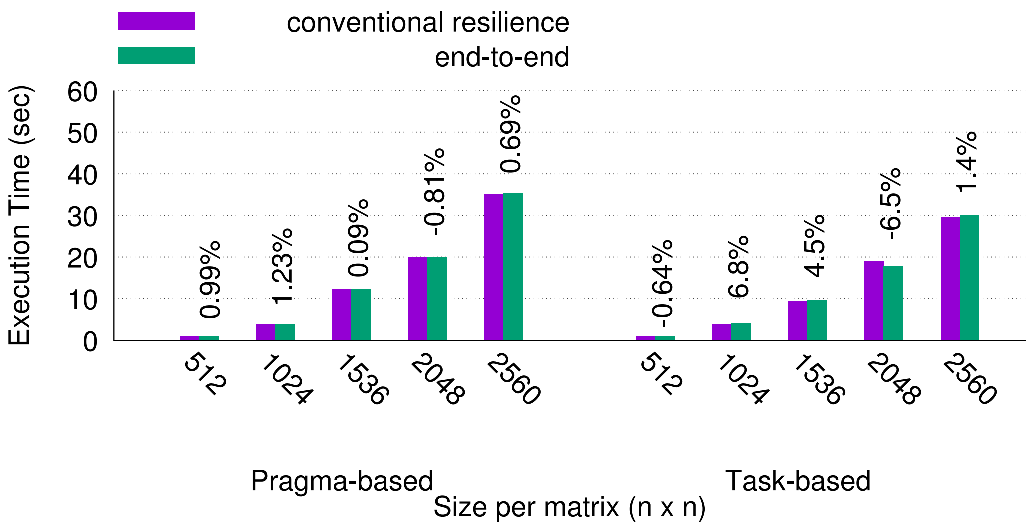


Figure 6.4 Overhead of fault-free execution of pragma-based and task-based matrix multiplication for end-to-end resilience vs. conventional resilience over 100 runs

First, we evaluate the performance overhead of end-to-end pragma-based and task-based matrix multiplication for fault-free execution and compare it with conventional resilience. The results are presented in Fig. 6.4. We can see that for fault-free execution, the overhead of end-to-end resilience over conventional resilience is negligible. In other words, we incur the same cost for end-to-end resilience with protection across kernels as conventional resilience, which just protects single kernels.

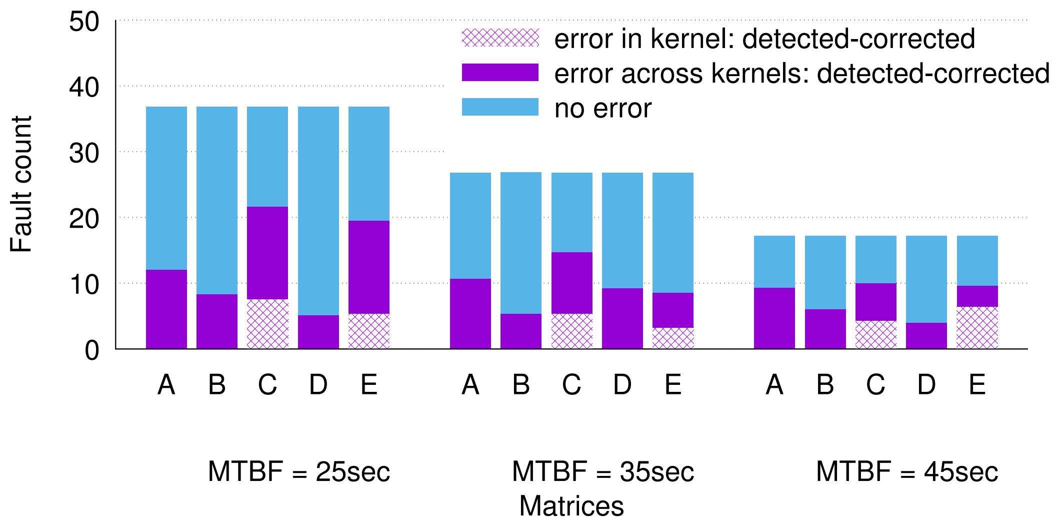


Figure 6.5 Fault injection in pragma-based matrix multiplication over 100 runs

Next, we evaluate the effect of fault injections into the matrices during execution using the fault injection mechanism that we have previously described. We inject faults at rates between 25 - 45 seconds for the pragma-based matrix multiplication with matrix sizes of 2560x2560. The results are shown in Fig. 6.5. The top-most bars represent injections into data structures that do not result in an error. This may be due to injections that happen before the data structure is loaded/initialized or injections that happened after the data was last used. The shaded regions for matrices C and E are faults that can be corrected using conventional resilience. These denote faults due to injections inside the kernel. The bars in purple represents faults that can only be corrected by end-to-end resilience as these faults are injected across kernels. Therefore, end-to-end resilience covers faults that occur both in the kernel and across kernels.

Similarly, we study the effect of injecting faults into the matrices for task-based matrix multiplication. Since the execution time of the task-based variant is lower, fewer number of faults are

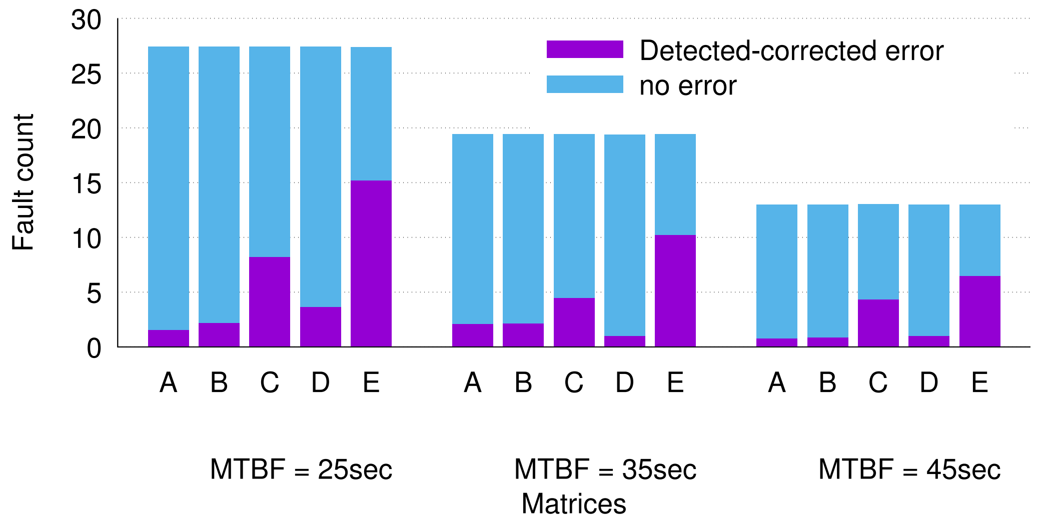


Figure 6.6 Fault injection in task-based matrix multiplication over 100 runs

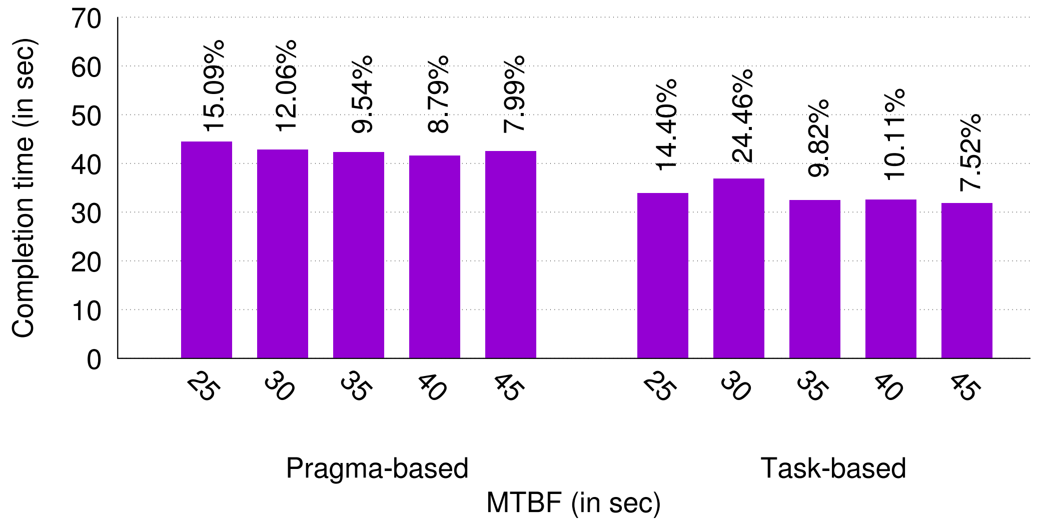


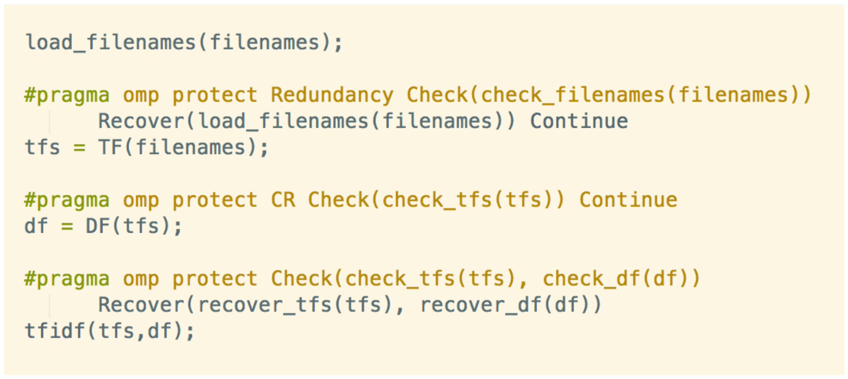
Figure 6.7 Completion time of pragma-based and task-based matrix multiplication with fault injections over 100 runs

injected at the same MTBF rate. Fig. 6.6 depicts the results of fault injection at rates between 25 - 45 seconds for matrices with size 2560x2560.

Finally, we compare the completion time of pragma-based and task-based matrix multiplication under different fault injection rates between 25 - 45 seconds in Fig. 6.7. We can see that the performance overhead for both the task-based and pragma-based variants is comparable. The task-based variant is faster in terms of absolute execution time because of more efficient cache reuse and lower overhead for the fine-grained corrections.

6.3 TF-IDF

We next assess the resilience capabilities of an MPI-based application, TF-IDF. TF-IDF is used to distinguish important terms from a large collection of text documents. It consists of two kernels, TF, which calculates the frequency of each term on a per-document basis, and DF, which calculates the total number of occurrences of each term. The final result is calculated as $tfidf = TF \times log\frac{N}{DF}$ for each term. We run the application on input sizes of 125MB, 250MB, 375MB and 500MB with 4 MPI ranks.



```
load_filenames(filenames);

#pragma omp protect Redundancy Check(check_filenames(filenames))
        Recover(load_filenames(filenames)) Continue
tfs = TF(filenames);

#pragma omp protect CR Check(check_tfs(tfs)) Continue
df = DF(tfs);

#pragma omp protect Check(check_tfs(tfs), check_df(df))
        Recover(recover_tfs(tfs), recover_df(df))
tfidf(tfs,df);
```

Figure 6.8 Code for TF-IDF along with the resilience pragma

Fig. 6.8 shows the steps of TF-IDF computation. Each rank first loads a subset of files. It then calculates the term frequencies from each of the documents and stores it in the `tfs` data structure. Then the document frequencies are calculated with `tfs` as input and `dfs` as output. Finally, the `tfidf` function calculates the respective values for each term.

6.3.1 Resilience Technique

For the TF-IDF application, we study the combination of redundant computation and checkpoint/recovery (CR). Each data structure has a checksum computed over its entire content. Redundancy is used to compare the values of these resulting checksums to confirm integrity. Similarly, CR is used to checkpoint the `tfs` data structure to persistent memory, and if we detect that a fault has occurred, we can restore the prior state from the stored checkpoint. In case of redundant execution, if we detect a fault, we reload the data structure from persistent memory. For our experiments, we do not execute a redundant process for the filenames data structure but instead compare the calculated and the stored checksums when we check the filenames data structure. This becomes feasible because of our fault injection method for TF-IDF, which is explained in a later section.

6.3.2 Pragma Expansion

Similar to pragma-based matrix multiplication, we enhance the TF-IDF application with pragmas in order to provide end-to-end resilience. For the code shown in Fig. 6.8, the Cetus transform pass generates the transformed code shown in Fig. 6.9.

```
load_filenames(filenames);
bool completed[3]={false};
while(!completed[2]){
  while(!completed[1]){
    while(!completed[0]){
      wake_up_shadow();  // primary wakes up shadow
      bool check[1];    // 1 check flag
      do{
        tfs = TF(filenames);
        if (!(check[0] = check_filenames(filenames)))
          if (!load_filenames(filenames))
            goto cleanup;
      }while(!check[0]);
      sleep_shadow();// primary puts shadow to sleep
      if (check[0])
        completed[0]=true;
    }
    Create_ckpt(tfs); // checkpoint "tfs"
    df = DF(tfs); // contains MPI calls
    if (1)
      completed[1]=true;
  }
   bool check[2]; // 2 check flags
   do{
    tfidf(tfs,df);
    if (!(check[0] = check_tfs(tfs)))
      if (!(recover_tfs(tfs))){
        completed[0]=false;//recompute region 0
        break;
      }
      if (!(check[1] = check_df(dfs)))
        if (!(recover_df(dfs))){
          completed[1]=false;//recompute region 1
          break;
        }
   }while(!check[0] && !check[1]);
   if (check[0] && check[1])
     completed[2] = true;
}
```

Figure 6.9 Code generated from pragmas for TF-IDF

6.3.3 Fault Injection

To inject faults into data structures in TF-IDF, we use a method similar to that of sequentially composed matrix multiplication. Even though the data structures are of vastly different sizes, we inject faults uniformly over the data structures so that we can assess the end-to-end resilience capabilities of the application. The normalized number of injections in each data structure based on their respective sizes for different values of MTBF is shown in Fig. 6.10. The injection handler selects a data structure randomly for injection. To finally inject a fault, the calculated checksum of the data structure is modified, which has the same effect as a bit flip in the corresponding data structure from the end-to-end resilience perspective. For the purposes of evaluation we inject faults into one of the MPI ranks.

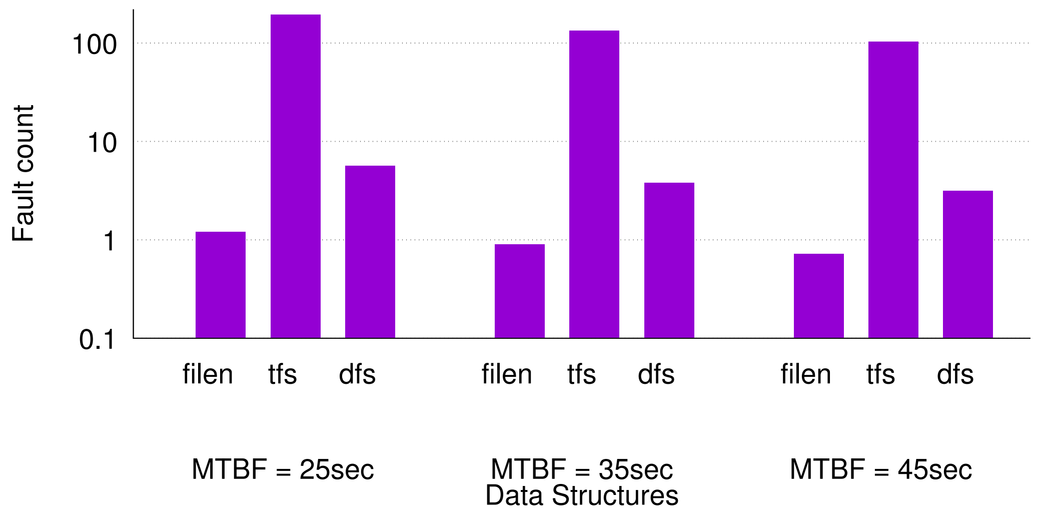


Figure 6.10 Normalized injection counts for each data structure over 100 runs

6.3.4 Performance Evaluation

We use 750 text books with a combined size of 500MB for the TF-IDF application with 4 MPI ranks. We evaluate the end-to-end resilience capabilities of the application with input sizes of 125MB, 250MB, 375MB and 500MB. Then we inject failures at mean rates of 25, 35 and 45 seconds to assess the performance under a fault injection scenario.

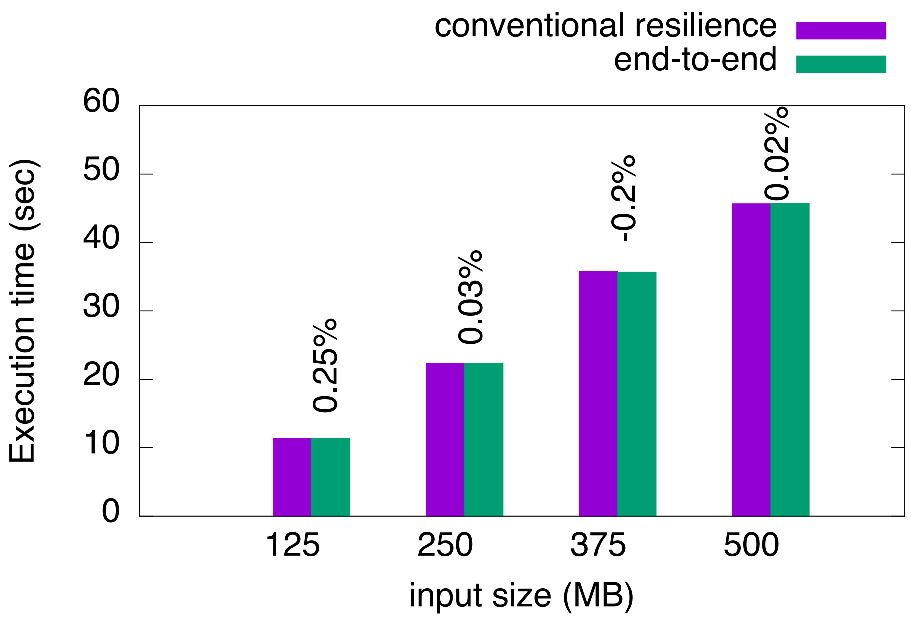


Figure 6.11 Overhead of fault-free execution of TF-IDF for end-to-end resilience vs conventional resilience over 100 runs

Fig. 6.11 shows the performance overhead of end-to-end resilience over conventional resilience for fault-free execution. Similar to matrix multiplication, the fault-free execution overhead for TF-IDF is also negligible. Therefore, there is no added cost for end-to-end resilience over conventional resilience that protects single kernels.

Finally, we evaluate the fault injection scenario with TF-IDF. We inject faults at rates of 25 - 45 seconds over 100 runs of the application. The results are presented in Fig. 6.12. The topmost bars represent injections that either affect data that has not been initialized yet or data that is no longer live. The shaded bars represent the faults that could have been corrected by conventional resilience as they occurred inside the kernel. The purple bars represent faults that are detected and corrected

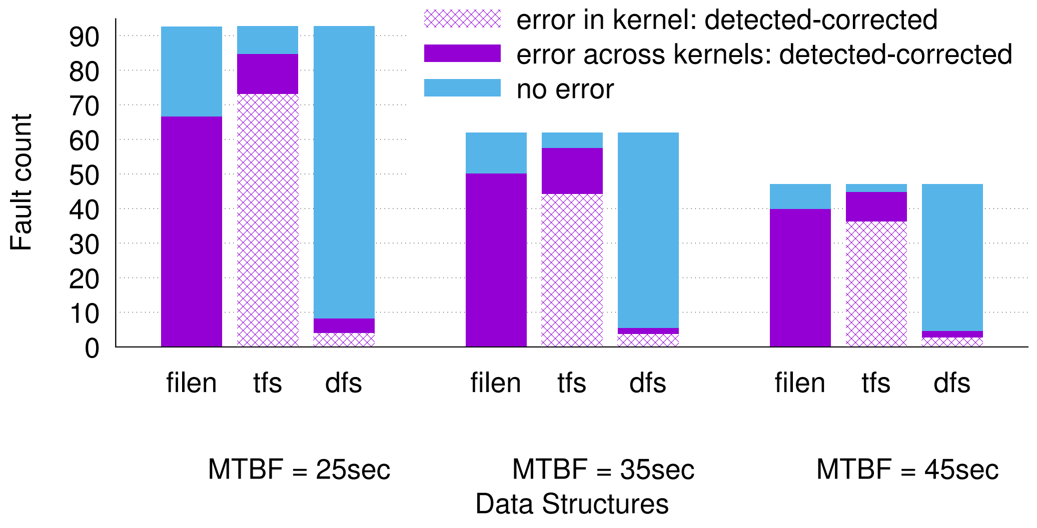


Figure 6.12 Fault injection in TF-IDF application over 100 runs

only by our end-to-end resilience technique as they span across kernels. End-to-end resilience can therefore detect and recover faults that would not have been detected at all by conventional resilience.

CHAPTER

7 LIMITATIONS AND FUTURE WORK

While our pragma scheme can extend protection boundaries across kernels to facilitate end-to-end resilience, there are certain limitations that an application programmer would need to be aware of. In this section, we discuss the limitations of the pragma-based approach for end-to-end resilience and also future work that can extend the utility of our technique.

One of the primary requirements for the pragma-based technique is that the computations be idempotent within the encapsulated regions i.e., if a kernel/region is called twice in a row, the results would not be different from it being called once during normal execution. If a computation were not idempotent, the burden to save and restore the global variables falls onto the programmer. Similarly, the application programmer needs to ensure I/O idem-potency of the application.

Secondly, the check and recover methods need to be selected on a per-kernel basis. While certain data structures can be made fault tolerant using ABFT, not all data structures have the properties necessary for ABFT. For other numeric solvers, we can use convergence tests as the method to check for faults. If, however, forward recovery is not possible, the application programmer would need to incorporate a recovery method that can restore the data from a checkpoint. We can default to a checksum over the entire data in case other checking methods are not available.

Finally, our technique protects certain data structures from soft faults, but cannot recover from faults in other parts of application memory, such as within code or control variables, which, when affected, could cause the application to crash or behave in an unpredictable manner. Protection of these aspects of memory are orthogonal to our work. Since a large fraction of HPC application memory is usually occupied by data structures, we can protect against a similarly large fraction of soft errors by protecting these data structures.

There are certain improvements to our technique that can further reduce the application programmer's burden for protection against soft faults. We can complement the compiler pass with live-variable analysis and capture/restore global or non-idempotent variables at region boundaries. We can also provide synergy between thread parallelism and resilience through the OpenMP library. This allows us to check and restore parts of a data structure on a per-thread basis rather than over the entire data structure.

CHAPTER

8 CONCLUSION

As HPC systems are becoming more complex and large, the number of faults these systems experience is projected to increase. Errors caused by bit flips in computational systems can severely restrict the progress of scientific applications. This makes resilience against such faults an important premise to ensure application progress for exascale systems. Applications often have different computational kernels that would benefit from their own specific protection schemes.

In this work, we design, implement and evaluate a resilience scheme to provide end-to-end protection for HPC applications against soft faults. Our pragma-based approach allows the application programmer to specify resilience methods on a per-kernel basis, while our compiler implementation provides end-to-end resilience for the entire application across kernels. We implement our pragma as a transformation pass using the source-to-source compilation capabilities of the Cetus compiler.

Finally, we evaluate the effectiveness of our end-to-end approach on two applications, sequentially composed matrix multiplication and TF-IDF. We inject faults at different rates into the application's data structures. We observe that our end-to-end resilience scheme recovers from these faults. We also show that end-to-end resilience can recover from more faults than conventional per-kernel resilience schemes, with negligible overhead of end-to-end resilience for fault-free executions.

BIBLIOGRAPHY

[AC06] Anderson, J. H. & Calandrino, J. M. “Parallel Task Scheduling on Multicore Platforms”. *SIGBED Rev.* **3**.1 (2006), pp. 1–6.

[And04] Andrews, M. “Instability of the proportional fair scheduling algorithm for HDR”. *Wireless Communications, IEEE Transactions on* **3**.5 (2004), pp. 1422–1426.

[Ash10] Ashby, S. et al. *The Opportunities and Challenges of Exascale Computing*. Tech. rep. U.S. Deptartment of Energy, 2010.

[Avi04] Avizienis, A. et al. “Basic concepts and taxonomy of dependable and secure computing”. *Dependable and Secure Computing, IEEE Transactions on* **1**.1 (2004), pp. 11–33.

[Bau05] Baumann, R. “Soft errors in advanced computer systems”. *IEEE Design & Test of Computers* **22**.3 (2005), pp. 258–266.

[BG11] Bautista-Gomez, L. et al. “FTI: High Performance Fault Tolerance Interface for Hybrid Systems”. *Supercomputing Conference*. 2011, 32:1–32:32.

[Bis11] Biswas, S. et al. “Exploiting Data Similarity to Reduce Memory Footprints”. *IEEE International Parallel & Distributed Processing Symposium*. 2011, pp. 152–163.

[Bri10] Brightwell, R. et al. “Transparent redundant computing with MPI”. EuroMPI’10. Stuttgart, Germany: Springer-Verlag, 2010, pp. 208–218.

[Cao15] Cao, C. et al. “Design for a soft error resilient dynamic task-based runtime”. *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*. IEEE. 2015, pp. 765–774.

[Cap09] Cappello, F. et al. “Toward Exascale Resilience”. *Int. J. High Perform. Comput. Appl.* **23**.4 (2009), pp. 374–388.

[Cap14] Cappello, F. et al. “Toward Exascale Resilience: 2014 update”. *Supercomputing frontiers and innovations* **1**.1 (2014).

[Che09] Chekuri, C. et al. “Minimizing Maximum Response Time and Delay Factor in Broadcast Scheduling”. English. *Algorithms - ESA 2009*. Ed. by Fiat, A. & Sanders, P. Vol. 5757. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2009, pp. 444–455.

[Che12] Chekuri, C. et al. “Online Scheduling to Minimize Maximum Response Time and Maximum Delay Factor”. *Theory of Computing* **8**.7 (2012), pp. 165–195.

[Che07] Chen, S. et al. “Scheduling Threads for Constructive Cache Sharing on CMPs”. *Proceedings Symposium on Parallel Algorithms and Architectures*. 2007, pp. 105–115.

[CD08] Chen, Z. & Dongarra, J. “Algorithm-Based Fault Tolerance for Fail-Stop Failures”. *Parallel and Distributed Systems, IEEE Transactions on* **19**.12 (2008), pp. 1628–1641.

[CW14] Chen, Z. & Wu, P. “Fail-Stop Failure Algorithm-Based Fault Tolerance for Cholesky Decomposition”. *IEEE Transactions on Parallel and Distributed Systems* **99**.PrePrints (2014), p. 1.

[Che16] Cheng, E. et al. “CLEAR: Cross-Layer Exploration for Architecting Resilience” (2016).

[Chu12] Chung, J. et al. “Containment Domains: A Scalable, Efficient, and Flexible Resilience Scheme for Exascale Systems”. *Supercomputing Conference*. 2012, 58:1–58:11.

[Cot10] Cotroneo, D. et al. “Software Aging Analysis of the Linux Operating System”. ISSRE ’10. IEEE Computer Society, 2010, pp. 71–80.

[Dav09] Dave, C. et al. “Cetus: A source-to-source compiler infrastructure for multicores”. *Computer* **42**.12 (2009), pp. 36–42.

[DÏ2] Döbel, B. et al. “Operating System Support for Redundant Multithreading”. *International Conference on Embedded Software*. 2012, pp. 83–92.

[Du12] Du, P. et al. “Algorithm-based Fault Tolerance for Dense Matrix Factorizations”. *Symposium on Principles and Practice of Parallel Programming*. 2012, pp. 225–234.

[Due03] Duell, J. *The design and implementation of Berkeley Labs Linux Checkpoint/Restart*. Technical Report. Lawrence Berkeley National Laboratory, 2003.

[Ell12] Elliott, J. et al. “Combining Partial Redundancy and Checkpointing for HPC”. *International Conference on Distributed Computing Systems*. 2012.

[Ell14a] Elliott, J. et al. “Evaluating the Impact of SDC on the GMRES Iterative Solver”. *International Parallel and Distributed Processing Symposium*. 2014, pp. 1193–1202.

[Ell14b] Elliott, J. et al. “Resilience in numerical methods: a position on fault models and methodologies”. *arXiv preprint arXiv:1401.3013* (2014).

[EB11] Engelmann, C. & Böhm, S. “Redundant Execution of HPC Applications with MR-MPI”. *International Conference on Parallel and Distributed Computing and Networks*. 2011, pp. 31–38.

[Fer12] Ferreira, K. “Keeping Checkpointing Viable for Exascale Systems”. PhD thesis. Albuquerque: Universoty of New Mexico, 2012.

[Fer11] Ferreira, K. et al. “Evaluating the viability of process replication reliability for exascale systems”. SC ’11. Seattle, Washington: ACM, 2011, 44:1–44:12.

[Fia12a] Fiala, D. et al. “Detection and Correction of Silent Data Corruption for Large-Scale High-Performance Computing”. *Supercomputing Conference*. 2012.

[Fia12b] Fiala, D. et al. *Detection and Correction of Silent Data Corruption for Large-Scale High-Performance Computing*. Tech. rep. 2012-5. Dept. of Computer Science, NCSU, 2012.

[Fia16] Fiala, D. et al. “FlipSphere: A software-based DRAM error detection and correction library for HPC”. *Distributed Simulation and Real Time Applications (DS-RT), 2016 IEEE/ACM 20th International Symposium on*. IEEE. 2016, pp. 19–28.

[Gei16] Geist, A. “How To Kill A Supercomputer: Dirty Power, Cosmic Rays, and Bad Solder”. *IEEE Spectrum* (2016).

[HA84] Huang, K.-H. & Abraham, J. “Algorithm-Based Fault Tolerance for Matrix Operations”. *Computers, IEEE Transactions on* **C-33**.6 (1984), pp. 518–528.

[Isl12] Islam, T. Z. et al. “McrEngine: A Scalable Checkpointing System Using Data-aware Aggregation and Compression”. *Supercomputing Conference*. 2012, 17:1–17:11.

[Joh90] Johnson, D. B. “Distributed System Fault Tolerance Using Message Logging and Checkpointing”. AAI9110983. PhD thesis. Houston, TX, USA, 1990.

[KK93] Kale, L. V. & Krishnan, S. “CHARM++: A Portable Concurrent Object Oriented System Based On C++”. *Conference on Object Oriented Programming Systems, Languages and Applications*. 1993, pp. 91–108.

[Kel10] Kelin, L. H.-H. et al. “LEAP: Layout design through error-aware transistor positioning for soft-error resilient sequential cell design”. *Reliability Physics Symposium (IRPS), 2010 IEEE International*. IEEE. 2010, pp. 203–212.

[Kic97] Kiczales, G. et al. “Aspect-oriented programming”. *Proceedings European Conference on Object-Oriented Programming*. Springer-Verlag, 1997, pp. 220–242.

[Kim14] Kim, Y. et al. “Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors”. *ACM SIGARCH Computer Architecture News*. Vol. 42. 3. IEEE Press. 2014, pp. 361–372.

[Lee09] Lee, S.-B. et al. “Proportional Fair Frequency-Domain Packet Scheduling for 3GPP LTE Uplink”. *INFOCOM 2009, IEEE*. 2009, pp. 2611–2615.

[Leo05] Leonardi, E. et al. “Joint optimal scheduling and routing for maximum network throughput”. *INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings IEEE*. Vol. 2. 2005, 819–830 vol. 2.

[Mar14] Marathe, A. et al. “Exploiting Redundancy for Cost-effective, Time-constrained Execution of HPC Applications on Amazon EC2”. HPDC ’14. Vancouver, BC, Canada: ACM, 2014, pp. 279–290.

[Mar03] Marsan, M. et al. “Multicast traffic in input-queued switches: optimal scheduling and maximum throughput”. *Networking, IEEE/ACM Transactions on* **11**.3 (2003), pp. 465–477.

[Mit05] Mitra, S. et al. “Robust system design with built-in soft-error resilience”. *Computer* **38**.2 (2005), pp. 43–52.

[Mit07] Mitra, S. et al. “Built-in soft error resilience for robust system design”. *Integrated Circuit Design and Technology, 2007. ICICDT’07. IEEE International Conference on*. IEEE. 2007, pp. 1–6.

[Moo10a] Moody, A. et al. “Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System”. *Supercomputing Conference*. 2010, pp. 1–11.

[Moo10b] Moody, A. et al. “Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System”. *Supercomputing Conference*. 2010, pp. 1–11.

[Nig11] Nightingale, E. B. et al. “Cycles, Cells and Platters: An Empirical Analysisof Hardware Failures on a Million Consumer PCs”. EuroSys ’11. Salzburg, Austria: ACM, 2011, pp. 343–356.

[PS07] Panzer-Steindel, B. *Data Integrity*. Tech. rep. 1.3. Switzerland: CERN, 2007.

[San08] Sanda, P. N. et al. “Soft-error resilience of the IBM POWER6 processor”. *IBM Journal of Research and Development* **52**.3 (2008), pp. 275–284.

[Sat12] Sato, K. et al. “Design and Modeling of a Non-Blocking Checkpointing System”. *Supercomputing Conference*. 2012.

[SG06] Schroeder, B. & Gibson, G. A. “A Large-scale Study of Failures in High-performance Computing Systems”. *Proceedings Conference on Dependable Systems and Networks*. 2006, pp. 249–258.

[Sch09] Schroeder, B. et al. “DRAM Errors in the Wild: A Large-scale Field Study”. *SIGMETRICS Perform. Eval. Rev.* **37**.1 (2009), pp. 193–204.

[Sha12] Shantharam, M. et al. “Fault Tolerant Preconditioned Conjugate Gradient for Sparse Linear System Solution”. *Supercomputing Conference*. 2012, pp. 69–78.

[Sni13] Snir, M. et al. “Addressing Failures in Exascale Computing”. *International Journal of High Performance Computing* (2013).

[Sri15] Sridharan, V. et al. “Memory Errors in Modern Systems: The Good, The Bad, and The Ugly”. *International Conference on Architectural Support for Programming Languages and Operating Systems*. 2015, pp. 297–310.

[Zhe12] Zheng, G. et al. “A Scalable Double In-memory Checkpoint and Restart Scheme towards Exascale”. *Workshop on FaultTolerance for HPC at Extreme Scale (FTXS)*. 2012.

[Zhe14] Zheng, Z. et al. “Fault Tolerance in an Inner-Outer Solver: A GVR-Enabled Case Study”. English. *High Performance Computing for Computational Science – VECPAR 2014*. Ed. by Michel et al. Vol. 8969. Lecture Notes in Computer Science. Springer International Publishing, 2014, pp. 124–132.