RUNTIME SPECULATIVE SOFTWARE-ONLY FAULT TOLERANCE

YUN ZHANG

A DISSERTATION
PRESENTED TO THE FACULTY
OF PRINCETON UNIVERSITY
IN CANDIDACY FOR THE DEGREE
OF DOCTOR OF PHILOSOPHY

RECOMMENDED FOR ACCEPTANCE
BY THE DEPARTMENT OF
COMPUTER SCIENCE

ADVISER: PROFESSOR DAVID I. AUGUST

JUNE 2012

Abstract

Transient faults are emerging as a critical reliability concern for modern microprocessors. Recently, microprocessors have been designed with lower voltage level ,smaller and faster transistors enabled by improved fabrication technology. A combination of increased density of transistors on chip, reduced noise margin of each transistor, and voltage scaling are making hardware systems more susceptible to transient faults than ever.

Both hardware or software solutions have been proposed for transient fault tolerance. The hardware approach typically adds redundant hardware modules to the system, thus requiring extra chip area as well as higher hardware design and verification cost. In addition, the scope and mechanism of fault tolerance are hardwired at design time, which could be suboptimal with the change of deployment environment. Unlike hardware solutions, software-only techniques do not require any specialized hardware extensions and are more flexible with the scope of protection and the change of environment. However, even the best-performing software-only fault tolerance techniques incur significant performance cost. The overhead of prior work comes from doubled register usage, frequent inter-core communication, or barrier synchronizations. These factors prevent existing software techniques from being adopted widely.

To address these problems, this dissertation proposes Runtime Software-only Speculative Fault Tolerance (RSFT). The key insights behind this dissertation are: (1) not all values are equally important. Transient faults may alter a transistor's value, which is never used. Only the values that will affect the externally visible behavior of a program must be verified before being used; (2) Value speculation can efficiently remove data dependences introduced by cross checking values produced in the program and its redundant copy with high confidence, thus significantly improves program runtime performance.

RSFT serves as a virtual layer between the application and the underlying platform.

It takes a program binary and designated execution arguments as input, and automatically creates two symmetric program instances for redundant execution, to utilize extra cores in a multi-core system. RSFT detects transient faults at system calls level in a non-invasive way, and exploits high-confidence value speculation to achieve low runtime overhead. Light-weight runtime checkpointing and background validation work together to provide transient fault recovery with only 6.17% overhead. The prototype of this framework was implemented and evaluated on a commodity multi-core system. The evaluation demonstrated that with this framework, transient fault tolerance can achieve best-in-class performance, full fault coverage, and fast recovery with no hardware module involved.

Acknowledgments

First, I would like to thank my advisor David I. August for his guidance throughout my years in graduate school. His passion for research and his vision on research directions always give me great inspiration. David gave me priceless suggestions continuously on research, study and life, which I am and will always be grateful for. During the years I worked in his research lab, I learnt from him not only how to conduct research, how to write scientific papers, but also how to think and act like a real researcher. I am thankful for the opportunities and challenges he gave me all these years, which made this dissertation possible. I could not have made my way today without his encourage and support. I believe I will continue to benefit from the knowledge he taught me during my study in Princeton.

I thank Prof. David Walker and Prof. Scott Mahlke for reading this dissertation extensively and providing insightful comments. The dissertation was significantly improved based on their feedbacks. I would like to also thank Prof. Jennifer Rexford and Prof. Doug Clark for serving as my thesis committee members. Their feedbacks that helped me to polish and refine my thesis.

This dissertation would not have existed without the help and support from everyone in the Liberty Research Group. We engaged in numerous brainstormings, discussions, presentations throughout the years in graduate school. The collaborative environment of this group is extremely healthy and helpful. I have found great friendship with the Liberty group members. I will never forget the sleepless nights we spent together before paper deadlines and the coffee we shared.

I would like to first thank the senior Liberty group memebers, Neil Vachharajani, Matthew Bridges, and Guilherme Ottoni for their guide and help in my early years into my PhD study. I must also thank Thomas Jablin, for his intelligence throughout my years in Princeton. His broad knowledge in almost everything was certainly both inspiring and educational. Without the discussions we had, this work would not have been possible. I am also indebted to Deep, who helped me on most of my paper submissions and magically made them better in every sense.

I also want to thank Prakash Prabhu, Souymadeep Ghosh, Jae W. Lee, Jialu Huang, Hanjun Kim, Nick Johnson, and Arun Raman, for making my life in graduate school an unforgettable experience. I will never forget the witty jokes Prakash tells everyday, the Bourburn shared with Nick when we are working for a paper deadline at midnight, and the countless lunch conversations with Jialu, and her talent in drawing.

During the years in Princeton, I had the luck to have great administrative support from the department and the Princeton University. In particular, my thanks go to Melissa Lawson, for handling my travelling, funding and many other things throughout the years. She made my life in Princeton a lot easier, which allowed me to focus on my research.

I also want to acknowledge the support from various organizations, who supported my PhD study. My work in Princeton was supported by DARPA, through contract FA8750-10-2-0253, the National Science Foundation through Grant No. 0627650 and 1047879, and the Gigascale Systems Research Focus Center, one of five research centers funded under the Focus Center Research Program, a Semiconductor Research Corporation program.

I would also like to thank my parents for their unconditonal love and support, since I was a child. As the only child in the family, it was difficult for them to accept the fact that I have to spend so many years abroad studying, far away from home. But they supported me and my dream without reservation. They always encourage me to pursue what I love and are pround of my accomplishments. I cannot image how much they sacrificed to allow me to achieve what I have today. Everything I achieved today, is truly theirs.

Last, but not the least, I want to thank my husband Yunpeng Wang, for his love, care, and company during my last several years in graduate school. We met each other when I was failing in almost every aspect of my life. He brightens my life with humor, love and great patience. I wish I could be half as witty and funny on my best days, as he is on his worst. I want to thank him for his unconditional support during highs and lows in our life together.

Contents

List of Figures

Chapter 1

Introduction

Reliability is one of the more critical concerns of a computer system. Computations on a hardware system are supposed to be calculating the correct value, as fast as possible. However, computer systems may fail due to hardware errors or transient faults. Transient faults, also known as soft errors, are caused by either environmental events, such as particle strikes, or fluctuating power supply, and are nearly impossible to reproduce. Transient faults are not necessarily attributed to design flaws and occur randomly after deployment. These faults do not cause permanent hardware damage, but may result in a complete system failure or data corruption.

In 1978, Intel Corporation first reported transient faults occurrence when its chip packaging modules were contaminated by uranium from a mine nearby. Hewlett Packard reported that the servers in Los Alamos National Laboratory were frequently crashing from transient faults resulted from cosmic ray strikes [28]. IBM S/390 [51], Boeing 777 airplanes [65], and HP's Himalaya [18] all incorporate redundant hardware for fault detection and recovery.

While transient faults are already a concern of modern computer systems, they will become a bigger problem in the future generations of architectures. As semiconductor technology continues to scale, the number of transistors on a single chip grows exponentially. As the chip area remains relatively constant, the density of transistors on a single chip is increased significantly. While the increasing number of transistors benefits processor performance, Increasing density of chips increases the chance that a particle strike affecting one transistor. Furthermore, the exponential reduction in transistor size and reduced noise margin of each transistor make them even less reliable. Moreover, extreme demands for energy efficiency drive aggressive voltage scaling, which leads to an even lower noise margin therefore less reliability. All of these technology trends make processor chips more susceptible to transient faults than ever before.

Due to the emerging requirement for reliable systems, transient fault tolerance has become one of the critical concerns in the semiconductor industry. A more recent study shows that a BlueGene/L machine with 104 nodes deployed in Lawrence Livermore National Labs experiences soft errors once every four hours [9]. For fast and reliable computation, it is critical to find efficient and low-overhead transient fault tolerance solutions for modern and future architectures.

1.1 Limitations of Existing Fault Tolerance Techniques

Due to the nature of transient faults, the common practice for fault tolerance is redundancy. Either the hardware module itself, or the computation on top of the hardware, is duplicated and compared against each other to verify the correctness of values. There have been two approaches, hardware and software. The hardware approach either duplicates hardware, or uses additional hardware to duplicate software execution, and the software approach duplicates program execution on the system.

Hardware solutions usually introduce extra hardware components, typically specialized for processors or storage systems. For example, caches and memory subsystems include extra information of error-correcting codes (ECC) to allow hardware checking for transient faults and recover at runtime. These bit-level techniques typically can protect values in memory subsystems, such as caches or main memory, against transient faults. But they are prohibitively expensive to be applied to the processors, due to the nature of frequent data updating in processors. Previous work on protecting register files using ECC is extremely costly in terms of both performance [57] and power [38]. This approach cannot be applied to, for example Arithmetic Logic Unit (ALU), without paying a significant penalty in chip area, power consumption or performance.

To solve the cost problem of ECC protection for processors, semiconductor industry has introduced redundancy for processor cores or hardware contexts [15, 18, 30, 41, 46, 47, 51, 61] to provide transient fault tolerance. Compared with the ECC or parity approaches, this hardware redundancy does not have to validate data at each computation and update parity information. These techniques can validate less frequently and reduces extra hardware required for error correction. This is commonly deployed in reliability-critical systems, especially server systems, such as the Compaq NonStop Himalya, IBM S390, and Boeing 777.

However, all hardware approaches involve extra hardware components, thus adding higher cost at design and validation time. In addition, the scope and mechanism of protection are hardwired at design time under an assumed failure model (e.g. single event upset model), and working environment (e.g. reference altitude), which may be suboptimal depending on deployment environments. Some hybrid techniques combine custom hardware extension and software redundancy [41, 60] for fault detection. Because of the hardware extensions, these approaches have the same limitations as the hardware techniques.

Current architectural trends toward multicore microprocessors naturally provide additional computing resources, thus making software redundant execution more viable than ever. Existing software proposals [35, 44, 49, 60, 67] typically insert redundant code into a program at compile time or runtime, and check for transient faults at runtime. Among these proposals, compiler-based techniques [44, 49, 60, 67, 12] are only applicable to programs whose source codes are available. Separately compiled modules, such as libraries, cannot be protected using compiler-based techniques due to the absence of source code at compile time. Additionally, these techniques are not applicable to legacy binaries, due to the absence of original source code, or compiler compatibility problems. Runtime techniques, such as [49], use dynamic instrumentation to instrument program binaries for fault detection at runtime. But this approach still has high performance overhead due to the cost of dynamic binary instrumentation, as well as frequent program synchronizations.

1.2 Research Objectives and Contributions

To overcome the cost, performance and applicability limitations of the previously proposed techniques, this dissertation introduces Runtime Speculative Software-only Fault Tolerance(RSFT), a comprehensive framework that efficiently protect program execution against transient faults without requiring any specialized hardware or program source code. It works on off-the-shelf hardware platforms and legacy program binaries.

This dissertation achieves its objective by taking advantage of the following insights:

- Not all values are equally important. Some register values may never affect the output of a program or change the control flow of a program. Applications' inherent fault tolerance enables RSFT to use this information to remove unnecessary value validation code to minimize the runtime validation and communication cost.

- Cross checking values produced from the program and its redundant copy creates data dependences between these two instances, and introduces barrier synchronization. Value speculation with high confidence during program execution can efficiently remove barrier synchronizations between hardware contexts, and significantly improve program performance.

With the above two observations, RSFT speculates results of predicatable system calls, so that the original program does not have to synchronize with the redundant copy(ies) to confirm the return values. This allows maximum overlapping of program execution, inter-thread/process communication, IO operations and transient fault detection.

This dissertation first presents a transient fault detection technique that provides efficient fault detection during program runtime without requiring program modification or recompilation. RSFT-Detection is a non-invasive speculative runtime system that detects transient faults. RSFT-Detection serves as a light-weight virtual layer between an application and the underlying platform. It takes a program binary as input, and automatically executes the binary redundantly using a process monitoring tool provided by the operating system to trap every system call. The arguments of the system calls invoked from both program instances are compared for correctness. A value mismatch means a transient fault has occurred and RSFT reports this to the user. Unlike some compiler-enabled techniques that must obtain knowledge of library functions for fault detection, RSFT must only understand the relatively stable and well-defined set of system calls.

To recover from faulty program execution and continue protecting execution from transient faults after recovery, this dissertation also proposes a comprehensive and efficient fault recovery scheme, named RSFT-Recovery. The original program and its redundant copy each creates a checkpoint of themselves at the beginning of the program, and keep creating checkpoints periodically during execution. The runtime light-weight checkpointing is created using system call `fork`'s copy-on-write provided by the operating system. One checkpoint includes two newly forked processes that are stalled for future use, and their register files. Once a checkpoint is created, the two stalled processes are probed by another background validation process. Their register files as well as memory images are compared against each other for validation. If everything is identical, this checkpoint is considered a **valid** checkpoint, otherwise it is **invalid** and is discarded. At program runtime, if a transient fault is detected, the previous valid checkpoint is used to resume correct program execution and the faulty ones are discarded. If no valid checkpoint is available, the program rolls back to the very beginning and restarts execution redundantly.

Compare with hardware or hybrid fault tolerance approaches, RSFT provides comparable performance. However, hardware solutions can provide protection for the underlying operating systems, and a wider range of hardware modules. For mission-critical systems, such as a space shuttle, the cost associated with hardware redundancy is not a major concern, but the speed of fault recovery and the fault coverage is critical. In these cases, hardware solutions suit the fault tolerance purpose better than RSFT.

RSFT provides a cheap and effective software-only method to protect applications from transient faults, with full fault coverage. However, redundant execution costs extra resources to achieve the fault tolerance. For example, RSFT consumes twice as much memory and CPU cycles as the original unprotected program. RSFT performs best when applied to applications that are computation intensive, and on multi-core or distributed architecture where extra unused CPU resource is available. Scientific computation is a good example of a typical application that RSFT works best on. For heavy IO bound applications, RSFT may add considerable runtime overhead. In addition, as RSFT does not provide protection for execution out of an application's user space, RSFT has wider window of vulnerability for those applications. These programs are better protected using hardware solutions than software ones.

One type of IO bound application is the video/audio decoding applications, such as media players. These application have high tolerance to faults naturally, and require high processing throughput. The high fault coverage of RSFT is not necessary, and RSFT's runtime misspeculation cost is not appealing for these applications. Compared with RSFT, other software solutions, such as symptom-based techniques, will introduce lower overhead, and provide acceptable level of fault tolerance.

A prototype of RSFT is implemented and evaluated in this dissertation. To study the reliability of RSFT, a in-depth analysis on its window of vulnerability is also discussed and measured via simulated fault injection. The performance of RSFT with fault detection and recovery are separately evaluated on a commodity hardware system.

It must be noted that the implementation of RSFT in this thesis does not support multi-threaded programs. However, this implementation can be extended to support multi-threaded programs with deterministic outputs. For applications, where non-deterministic outputs are acceptable, RSFT may raise false alarms. Research on applying RSFT to these applications and redefining faulty behavior for programs with non-deterministic outputs remains a future work of this thesis.

The contributions of this thesis are:

- Design and implementation of RSFT, the fastest transient fault tolerance technique known to date. RSFT delivers full transient fault coverage with a geomean performance overhead of 6.17% for 23 SPEC CPU benchmarks using a commodity multi-core system.

- A detailed evaluation of the reliability of RSFT by injecting transient faults into both register files and memory space. This work is the first to simulate and evaluate the reliability of a software transient fault tolerance technique against memory faults.

- A transient fault tolerance framework that does not require any program modification and recompilation.

1.3 Dissertation Organization

Chapter 2 provides background information on transient faults, the state-of-the-art transient fault tolerance techniques and their limitations. Chapter 3 and Chapter 4 propose software-only speculative transient fault detection and runtime checkpointing and recovery techniques, respectively. Their design and implementation details are described in these chapters as well. Chapter 5 introduces the methods used for evaluation in this thesis, comparing with common practice in the field and provides rationale behind these metrics. Chapter 6 summaries a broad quantitative evaluation of RSFT on a commodity multi-core system. Finally, Chapter 8 concludes this dissertation and discusses future directions of research.

Chapter 2

Background

Transient faults are emerging as a critical reliability concern in microprocessors. These faults are usually caused by external events such as particle strikes, or internal events, such as power fluctuation [4, 34, 43, 48]. These events can cause additional charge to be deposited and therefore alter the value of a single transistor. Transient faults do not result in permanent hardware damage, but may lead to system failures or affect program execution.

2.1 Transient Faults Problem

Transient faults have already caused significant failures in deployed computer systems. Sun Microsystems, for one, acknowledges that customers such as America Online (AOL), eBay, and Los Alamos National Labs have experienced system failures due to transient faults [5]. In 2005, a 2048-CPU Hewlett Packard server system in Los Alamos National Laboratory was frequently crashing because of transient faults caused by cosmic ray strikes [28]. In addition to causing machine crashes, recent research has also shown that soft errors can lead to the derivation of secret keys in RSA public-key cryptography [3, 7] and induce security vulnerabilities [16].

Intel [54] and IBM [68] noted that radioactive materials can change the values in storage devices of processors. Manufacturing process has been taking caution to protect hardware from being exposed to radiation. While the fault rate per bit remains relatively constant over technology generations [17], exponentially growing transistor counts, combined with aggressive voltage scaling, make microprocessors more susceptible to transient faults than ever before. Additionally, cosmic radiation effects on modern microprocessors are hard to trace and evaluate because it changes with the system's deployment environment, such as altitude, geography and periodic solar phases. A majority of solar radiation is deflected from the planet surface by the Earth's magnetic field. The amount of radiation can vary significantly depending on the geographic location.

It is also difficult to measure the failure rate after deployment due to the unpredicatablity of transient faults. Previous research has shown that average failure rate per bit is between 0.01 to 0.001 per 10^9 hours[17, 22, 32, 56]. SPARC64 was reported to have 80% of its 200,000 latches covered by some form of fault protection [2].

Future generations of processors will be even more susceptible to transient faults. Competing factors for the reliability of individual bits in a processors suggest the failure rate per bit will remain constant for the next few generations [17, 22]. The reduction in transistor size decreases the probability that cosmic radiation will strike one single transistor, but each transistor also has a reduced critical voltage (the charge necessary to change its values), increasing the likelihood that a particle strike will affect the stored state. While the failure rate per bit is staying roughly constant, the failure rate per processor is increasing proportional to the number of transistors on a chip, which increases at an exponential rate.

Voltage scaling is also one factor that may increase the fault rate per transistor in the future. Active power consumption[1] of a chip has quadratic relationship with the supply voltage. As a result, the tightening power consumption constraint requires the aggressive

[1]Active power consumption is the total power consumed minus the leakage power

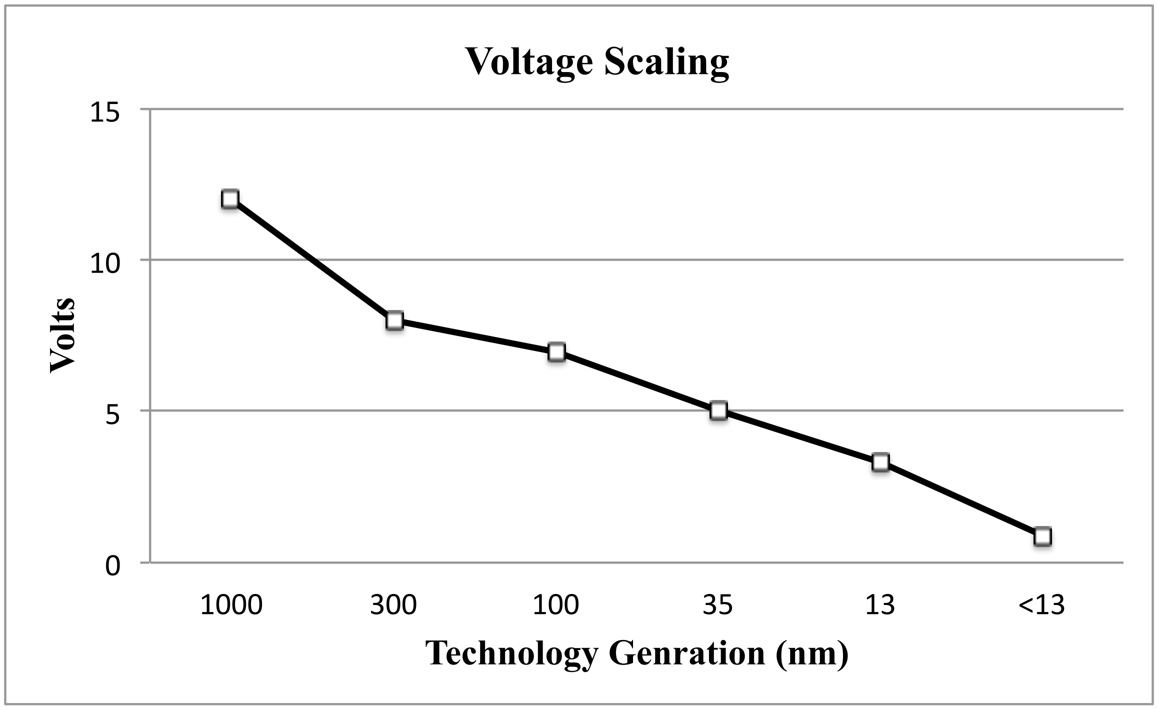


Figure 2.1: Voltage scaling over generations of micro-processors. * Data source: Intel

voltage scaling. Figure 2.1 shows the voltage scaling trend over generations of micro-processors [8]. This technology trend leads to reduced noise margins, which makes transistors more vulnerable against particle strikes and power fluctuation.

The soft error rate per chip, including processor logic and on-chip memory subsystems, is increasing with the decrease of technology node. As shown in Figure 2.2, processors produced with the 32nm technology is 50% more vulnerable than the ones made with 180nm [8]. As technology leans toward smalled logic units, the soft error rate will continue to scale up. In 2004, Borkar et al. estimates about 8% reliability degradation per bit per generation of microprocessors. It is getting more and more difficult to design and produce computer systems with reliable components today, and even so in the future.

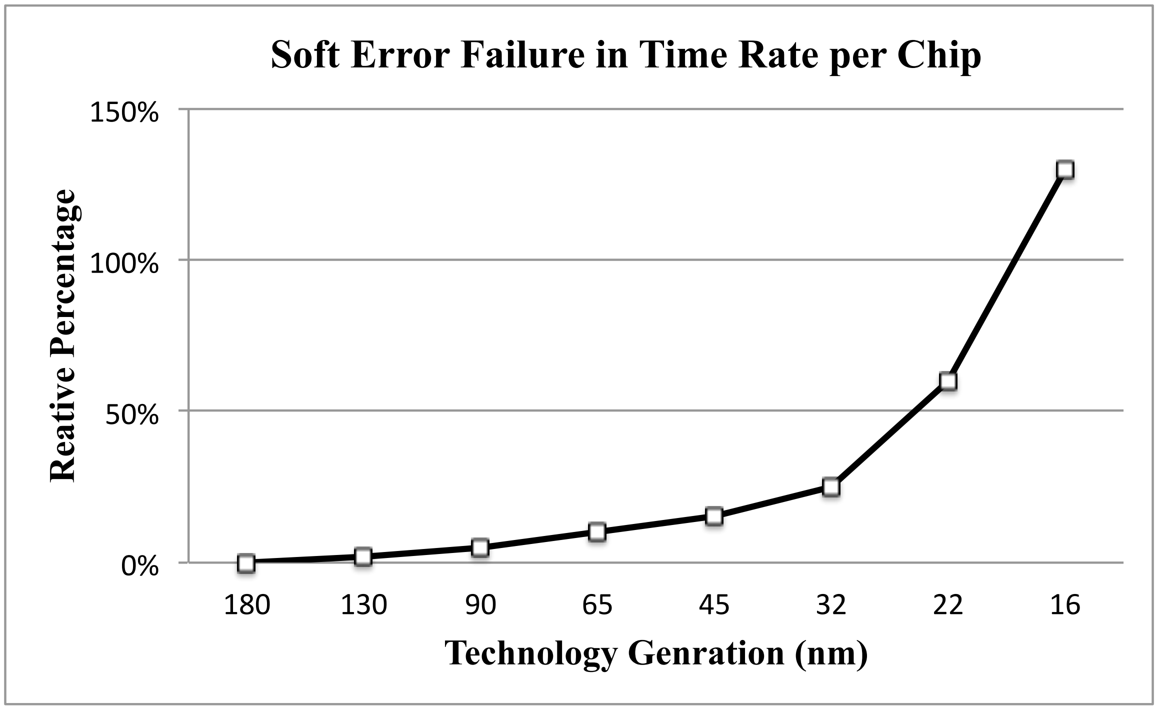


Figure 2.2: Technology trend and soft error failures in time. *Data source: Intel

2.2 Existing Methods for Transient Fault Tolerance

Fault tolerance solutions have been proposed to detect transient faults during program execution via redundant computation, and recovery from the faults. The sphere of replication (SoR) [41] is used to identify the scope of fault coverage. Values that enter the SoR must be replicated for redundancy and values that exit the SoR must be checked for faults to ensure their correctness. The implication of SoR is the scope of transient fault protection. All fault tolerance techniques proposed so far depends on redundancy, either in space or time. Table 2.1 lists most recent existing representative fault detection techniques.

2.2.1 Hardware Redundancy

Hardware redundancy provides transparent fault tolerance using extra specialized hardware, such as *watchdog* processor in [26]. In reality, IBM S/390 [51], Boeing 777 air-

Approach	Technique	Main Memory Usage	Need Source Code	Hardware Contexts		Sphere of Replication		Reported Overhead	
				Detection	Recovery	Processor	Memory	Detection	Recovery
Specialized Hardware	SWAT [25]	1×	No	2	2	Most	None	5%	
	AR-SMT [46]	1×	No	8	-	All	None	16.7%†	-
	CRT [30]	1×	No	2	-	Most	None	Unreported	-
	SRT [41]	1×	No	2	-	Most	None	Unreported	-
	Hybrid-SRMT [60]	1×	Yes	2	-	Most	None	19%†	-
Thread-local Duplication	EDDI [35]	2×	Yes	1	-	Most	All	52.2%	-
	SWIFT [44]	1×	Yes	1	-	Most	None	45%†	-
	Shoestring [12]	1×	Yes	1	-	Most	None	15.8%†	-
Redundant Multi-Threading	SRMT [60]	1×	Yes	2	-	Most	None	400%	-
	DAFT [67]	1×	Yes	2	-	Most	None	38%	-
Process-based Redundancy	PLR [49]	2×	No	2	3	Most	Some	16.9%	41%
	RSFT[This Thesis]	2×	No	2	2	Most	All	3.54%	6.17%

Table 2.1: Comparison Among Transient Fault Tolerance Techniques. † indicates the results was obtained on a simulator, not commodity hardware.

planes [65], and HP's Himalaya [18] all incorporate triple-redundant hardware for fault detection and recovery on-the-fly.

However, redundant execution in custom hardware can increase the transistor count of a processor by 20-30%. This also leads to extra chip area and verification cost [2, 51]. For example, AR-SMT [46], SRT [41], and CRT [30] all use at least one or more processing unit for transient fault detection only. Additionally, the scope and mechanism of protection are hardwired at design time under an assumed failure model (e.g. single event upset model), and working environment (e.g. reference altitude), which may be suboptimal depending on deployment environments.

One hardware solution, named SWAT [25], is a symptom-based fault tolerance technique. It detects transient faults by observing the program behavior at runtime using an assumed-to-be faulty-free processor. If a program crashes, SWAT will report it and recover from a previous checkpoint, which is also assumed to remain intact after the checkpoint is made. This approach also does not handle Byzantine failures [2], which is more difficult to detect than program crash.

Early multi-threaded fault detection techniques rely on specialized hardware to execute

[2]Byzantine failures means program runs and completes even it is affected by faults. Programs with this kind of failure does not crash, but will produce erroneous, random or even malicious results.

redundant copies of the program for transient fault detection and recovery. Rotenberg's AR-SMT [46] is the first technique to use simultaneous multi-threading for transient fault detection. An active thread (A) and a redundant thread (R) execute the same program at runtime, and their computation results are compared to detect transient faults. This method uses an 8-way simultaneous multi-threading trace processor to achieve its purpose. Simultaneous Redundant Threading (SRT) [41] and Chip-level Redundant Threading (CRT) [30] exploit simultaneous multi-threaded processors and multiple cores respectively for redundant execution and value checking. These techniques use duplicate hardware modules, and check values when they escape the SoR for fault detection. Limited by its specialized hardware requirement, these approaches are also not widely adopted.

For caches and memory subsystems, extra information of error-correcting codes (ECC) is a common practice to allow hardware checking for transient faults and recover at runtime. This bit-level technique typically protects values in memory subsystems, such as caches or main memory, by encoding the bits and verifies the value against transient faults. Although this is effective to tolerant single-bit-flip transient faults, they do not work well against multiple bit flip events. Additionally, they are prohibitively expensive to be applied to the processors, or on-chip caches, due to the nature of frequent data updating in processors. Previous work on protecting register files using ECC is extremely costly in terms of both performance [57] and power consumption [38]. Similar results have been found for on-chip cache ECC protection as well. Systems with ECC protected caches suffers nearly 2% performance degradation, and 15% chip area increase. With the increase of on-chip cache size and exponentially increasing transistor count of processors, protecting processors and on-chip caches means paying a significant penalty in chip area, power consumption, performance, or all of the above.

2.2.2 Software Redundancy

Compared with hardware redundancy approaches, software-only solutions are more appealing for its cost-efficiency and flexibility [15, 30, 44, 60, 35]. Although software solutions also use some extra resources for redundancy, they usually do not require specialized hardware modules. Instead, these techniques exploit redundancy using existing under-employed processors, register files or cores to achieve redundancy. Software transient fault detection techniques typically fall into three categories: thread-local duplication, redundant multi-threading and process-based redundancy, as shown in Table 2.1. Thread-local duplication techniques such as EDDI [35] and SWIFT [44] redundantly execute instructions within a single thread, exploiting instruction-level parallelism to improve performance. Shoestring [12] combines symptom-based fault detection with selective instruction duplication to achieve lower overhead than both EDDI and SWIFT, but with lower fault coverage. Redundant multi-threading techniques (e.g. SRMT [60] and DAFT [67]) use multiple threads to execute program codes redundantly. Process-based redundant techniques (e.g. PLR [49]) use multiple processes instead of threads, at the cost of maintaining multiple memory states.

All these techniques are typically implemented using either compiler transformations or runtime systems. Compiler-based approaches toward transient fault detection, such as EDDI [35], SWIFT [44], SRMT [60], DAFT [67], and Shoestring [12], all require program source code for recompilation, and cannot detect any transient fault occurring in separately-compiled modules.

EDDI [35] provides fault tolerance for programs via instruction duplication within the same thread. Each memory location has a corresponding shadow location in memory for duplicated instruction, which increases memory pressure. SWIFT [44] also exploits unused computing power of multiple-issue processors by duplicating program execution within

```
A: for (iter = 1; iter <= timesteps; iter++) {
B:    computeMatrix();                              int printf(const char *format, …) {
C:    printf("%d: %.2e %.2e %.2e\n",             D:   …
             Node[src], Matrix[pos][0],                  syscall(sys_write);
             Matrix[pos][1], Matrix[pos][2]);      E:  …
F:    updateValues();                                 }
   }
```

Figure 2.3: Simplified Code Example from SPECINT 2000 Benchmark 183.equake

the same thread. Achieving low runtime overhead for fault detection, SWIFT-transformed codes require twice as many registers, potentially causing register spills. In SWIFT, memory operations can only be performed once due to potential side effects. Transient fault checking instructions must be inserted before every memory operation. Unlike SWIFT, RSFT exploits multiple cores for redundant execution to minimize runtime overhead, and utilizes separate memory spaces provided by multiple processes to eliminate per-memory operation fault checking.

Software-based Redundant Multi-threading (SRMT) is a software solution that achieves redundancy with multiple threads. The SRMT techniques uses compiler transformation to automatically generate redundant code for runtime fault detection. However, due to the single memory state maintained during execution, redundancy is lost at memory operations. These techniques cannot issue redundant store instructions because only one shared memory state is maintained. Before a memory operation is executed, its operands are communicated between threads and checked for consistency. Consequently, frequent barrier synchronization is required and adds significant performance cost. When a real transient fault triggers an exception, SRMT invokes the program's exception handler to catch the fault, registering a false positive and possibly changing the program's behavior.

Compared with compiler-based techniques, runtime techniques do not require any program source code to be recompiled and can detect transient faults for separately compiled modules as well. One such implementation called PLR [49], a dynamic instrumentation technique, provides transient fault detection with the minimum runtime overhead (16.9%)

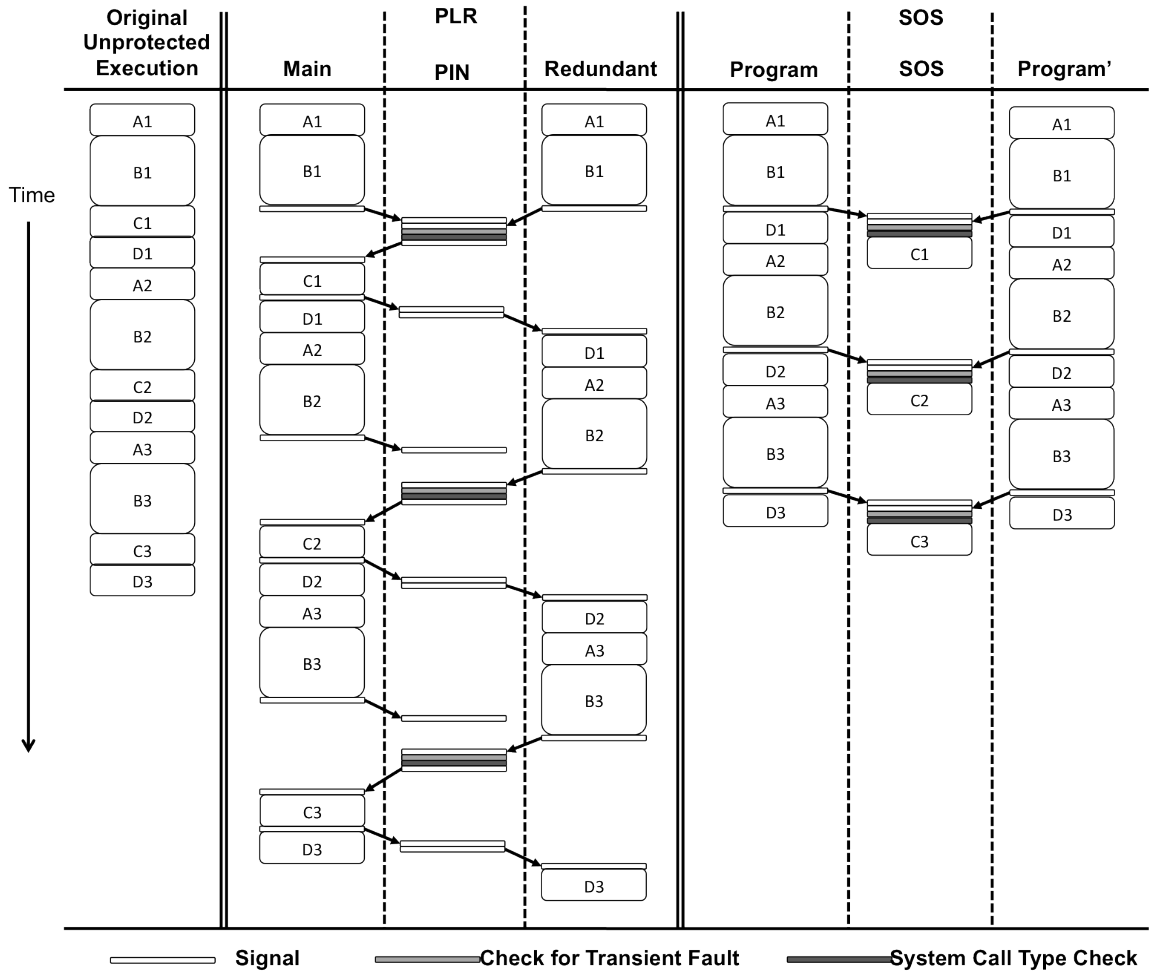


Figure 2.4: Execution plan of transient fault detection without and with speculation for example program in Figure 2.3 with `timesteps` = 3. The execution time of same instruction blocks, such as B1, B2, are slightly different across both processes because of various runtime factors such as cache behavior and process scheduling.

among all software solutions with full coverage. This technique duplicates the original program into several instances at runtime, maintaining one private memory space for each instance. Only externally visible values need to be verified before they escape user space.

PLR duplicates the execution of the application and its libraries, protecting program execution on both processors and memory, but not in the operating system. As a result, the arguments of a system call escape the SoR and must be checked before the system call is executed in kernel mode. Similarly, the return value of the system call is an input to the SoR, and needs to be replicated in both program instances. Therefore, PLR synchronizes the main and the redundant processes at every system call for transient fault detection and returns replicated value. The main process executes the system call. The redundant process only resumes execution after the system call is completed. This barrier synchronization puts inter-core communication on the critical path of program execution, leading to slower performance.

Other techniques, such as the pi bit [64] proposed by Weaver et al. and dependence-based checking [58] by Vijaykumar et al. detect faults that affect program behavior. This is done by following the propagation of faults through the entire program. For optimal performance, these techniques have to use herotic alias analysis to find the minimum set of dependences that fault tolerance code must track at runtime. RSFT achieves the same goal by only detecting non-benign faults only when they are about to affect program output. No program source code or alias analysis is needed to analyze the fault propagation ahead of time.

2.3 Limitations of Existing Techniques

Hardware redundant computing requires extra chip area, extra logic units, and additional hardware verification. The scope of protection and fault detection scheme are usually hardwired at design time, which limits the system's flexibility. On the other hand, software redundancy is more flexible and much cheaper in terms of physical resources. Software approach avoids expensive hardware and chip development costs. The ongoing multicore design provides increasing parallel resources in hardware, making software redundancy solutions more viable than ever.

Although software approach is more appealing, the significant runtime overhead of existing software techniques prohibits the previously proposed schemes from being deployed on commodity multi-core systems. For example, SRMT [60], a compiler-assisted redundant multi-threading implementation, adds 19% performance overhead, even with specialized hardware communication queue. PLR [49], as an example of runtime technique using process level redundant execution, adds 16.9% average runtime overhead for un-optimized programs, and much more overhead on IO intensive applications.

This dissertation proposes RSFT, the first speculative transient fault tolerance framework that provides the fastest transient fault detection and recovery. RSFT automatically duplicates program execution in a non-invasive way. As a result, RSFT maintains replicated memory states and has the SoR that covers both the processor cores and the memory subsystems. It eliminates frequent barrier synchronization via speculation, and provides efficient recovery mechanisms after a transient fault is detected. Compared with previous work, RSFT yields the lowest performance overhead for transient fault detection without compromising fault coverage.

To illustrate the advantage of RSFT , Figure 2.3 shows a simplified code example from 183.equake, a SPECFP 2000 benchmark program. Figure 2.4 compares non-speculative execution plan versus speculative execution plan in RSFT. These execution plans demonstrate that barrier synchronizations add considerable runtime overhead to the program. Although both process instances are executing the same program binary, the cycles spent on executing each piece of code, such as `B1, B2, ...`, are not the same because of various runtime factors such as cache behavior and process scheduling. Forcing barrier synchronization at every system call accumulates such timing difference, consequently slows down the whole program execution. In contrast, RSFT allows one process to speculate the return values of `sys_write` system call without actually executing it, therefore does not require waiting till the other process to invoke the same system call and the barrier synchronization between several program instances. If a misspeculation occurs, an efficient misspeculation recovery scheme is employed to continue execution from a previous verified program state. Combining all the features above, RSFT achieves very low runtime overhead with full fault coverage.

To understand and compare the performance of PLR, the best performing software transient fault tolerance technique, this thesis also implemented an approximation of PLR called RSFT-NoSpec. The implementation of RSFT-NoSpec faithful replicates the original paper, but using the light-weight processing monitoring tool utilized in RSFT , instead of PIN, the dynamic binary instrumentation employed in the original paper. As shown in Figure 2.1, RSFT-NoSpecis a lower-bound approximation of PLR with regard to the performance overhead.

Chapter 3

Runtime Speculative Transient Fault Detection

This chapter introduces the transient fault detection mechanism in RSFT , called RSFT-Detect [66]. Compared with previously proposed fault detection work, this technique does not require program source code, runtime instrumentation or program recompilation. RSFT-Detect is a fast and effective transient fault detection system that exploits the efficiency and functionality of an OS-level process monitoring tool `ptrace`. The `ptrace` utility is a POSIX standard that is provided by Linux/Unix, Mac OS, and Solaris systems to provide process monitoring and debugging capabilities. This kind of monitoring utility is exploited in RSFT-Detect as a method of trapping system calls and comparing the values of their arguments to detect transient faults. This approach ensures that RSFT-Detect can transparently detect transient faults occurring in a non-invasive way.

This chapter first discusses the existing research on transient fault detection, and compares RSFT-Detect with other alternative techniques. Section 3.2 describes the overall structure of RSFT-Detect . The following sections details the technique and its implementation. Section 3.11 discusses the window of vulnerability of RSFT-Detect, and compares it with previous works.

3.1 Related Work

Software-only transient fault detection techniques are advantageous compared with hardware approaches because of their flexibility and no extra hardware design cost. They are typically cheaper and easier to deploy on a commodity systems.

Among existing software transient fault detection techniques, many proposals use the compiler to recompile application code and insert proper instructions for transient fault detection. While these techniques can obtain a global view of the program code and instruction dependences to further optimize code for performance, they are limited by the need for the program source code. Separately compiled modules, such as existing libraries, often do not have source code available for recompilation. Thus, these separately compiled modules are a part of the window of vulnerability of these techniques. Previous compiler-based fault tolerance techniques do not evaluate their fault coverage when processors are running library code.

Another approach, being aware of such limitations, proposes to instrument program binary code at runtime for transient fault tolerance [49]. Although this approach protects program and its dynamically linked binaries, it suffers from high overhead incurred by the dynamic instrumentation. For example, Pin instrumentation framework, which was used in PLR [49], is reported to add 20-30% runtime overhead on average, just through executing a program binary within the framework. The evaluation in this dissertation runs all SPEC benchmark programs within the PIN dynamic instrumentation framework, and proves the estimates. The average performance overhead of PIN is 44.54% across all benchmark programs. The barrier synchronization in PLR further adds 17.27% more overhead on top of that.

3.2 Overview

Unlike all previous proposals, RSFT-Detect utilizes a process monitoring utility (`ptrace`) to intercept system calls invoked by a program. As a light-weight interface between the operating system and user applications, `ptrace` adds very little runtime overhead. Figure 3.1 demonstrates the overall structure of RSFT-Detect and the interaction between several components of the system. RSFT-Detect first takes the program binary and its input, then spawns one process that execute the binary redundantly. Upon process creation, RSFT-Detect immediately pauses that process, and inject a `fork` system call into the just-created child process `App`. The child process `App` spawns another process `App'` from itself, inheriting all its virtual address table and signal handling table. This is critical to eliminate potential false-positives, especially on systems with address space randomization enabled. From then on, `App` gives up its parent-ship of `App'` to the tracer process. Both `App` and `App'` becomes processes that are traced only by the tracer process RSFT-Detect.

During the execution, RSFT-Detect serves as a virtual layer between the application and the underlying OS services and devices. It traps every system call invoked by either program instance. After a system call is trapped, the process's register file is examined to find out the type of the system call. RSFT-Detect then compares the system call's arguments against those in the other program instance to check for transient faults, if available, according to its specific calling context. If the system call reads a process' memory space through pointer arguments, the memory content is also checked. If no transient fault is detected, RSFT-Detect executes the system call and lets the program instance continue execution. This runtime system predicts the results of system calls when possible, which allows speculative execution of the program. When misspeculation is detected after a system call is completed, misspeculation recovery schemes are employed to ensure continuous correct execution of both program instances.

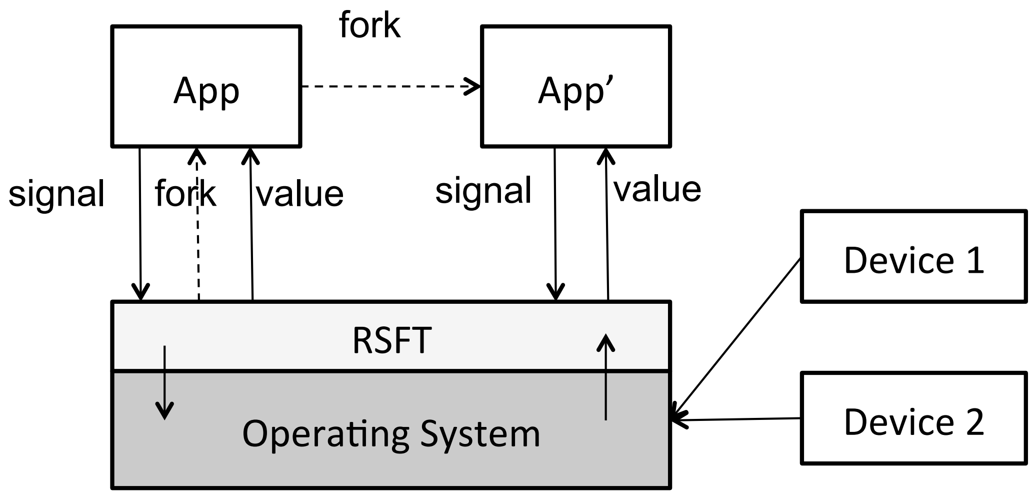


Figure 3.1: RSFT-Detect Structural Overview

3.3 Automatic Process Duplication

As shown in Figure 3.1, RSFT-Detect first fork a new process from itself, and starts executing the binary with its arguments in the newly create process. Algorithm 1 describes the detailed steps of how to force a child process `W1` to create a copy of itself and gives up its parent-ship to RSFT-Detect.

Algorithm 1 Non-invasive Process Duplication

```
1: W1.pid = fork()
2: copy original register file of W1
3: $W1.Regs.rax = SYS\_fork$
4: // the instruction SYSCALL is 0F 05, which takes 2 bytes
5: $W1.Regs.rip = W1.Regs.rip-2$
6: continue executing the system call
7: // program will re-enter interrupt and execute system call fork
8: wait for kernel finishes executing SYS_fork
9: $W2.pid = W1.Regs.rax$
10: copy back original register file of W2
11: continue executing W1 and W2
```

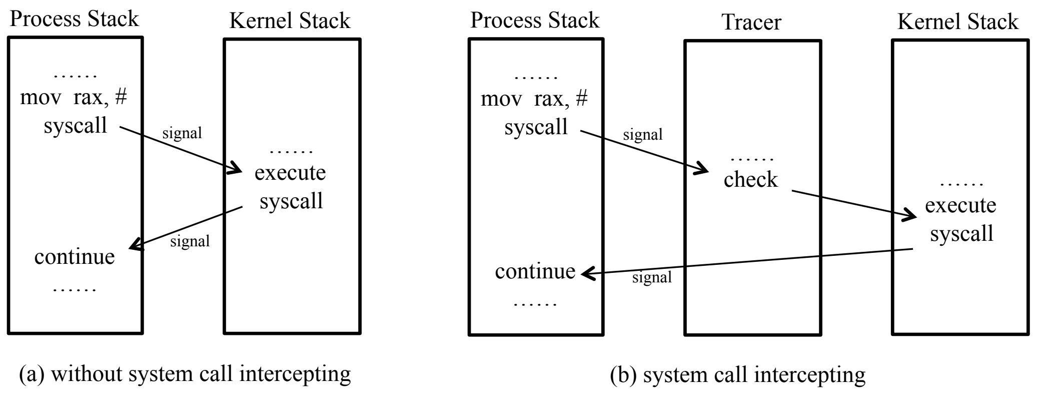


Figure 3.2: System Call Intercepting Mechanism

3.4 System Call Trapping

RSFT-Detect detects transient faults at system call interface level. The idea is to intercept all system calls initialized from two identical program instances. There is a fixed set of well-defined system calls provided by the operating system to applications. For modern Linux operating system, a total of 298 system calls are defined. It is reasonable to specify customized system call arguments verification for each system call.

RSFT-Detect serves as a thin layer between programs and the underlying system and hardware devices. Figure 3.2 illustrates the process of system call intercepting and argument verification process. Depending on the type of the arguments, different checking methods are used, including value comparison and memory content comparison. As the interface of system calls are very well defined, all system call arguments must be a register value. If an argument is a pointer, the size of the pointed to content must be specified in the system call arguments as well. Operating system itself also leverages this knowledge to handle the user space memory content. As a result, it is possible to manually inspect all system calls, and write speculation code for each system call by hand.

Despite the redundant execution of program binaries, side-effecting work should only happen once. Since system calls have well defined interfaces and calling contexts, it is

	Duplicate	Speculation	Example	Distribution in SPEC
Sync	No	No	sys_read(0), sys_readv(0), sys_lseek(0)	0.01%
Asynch	Yes	No	sys_mmap, sys_mremap, sys_brk	19.33%
	No	Yes	sys_write, sys_writev, sys_setitimer	80.66%

Table 3.1: Categorized System Calls

possible to manually examine all system calls and select out the side-effecting ones. For example, a `fprintf` (which transitively calls a `write` system call) can not be redundantly executed. Based on whether the results of the system call can be speculated and whether the system call needs to be executed in both copies of the program instances, all system calls are categorized into four categories. Table 3.1 gives the classification and some examples from existing system calls. The last column gives the distribution of system calls among these four categories, sampled from SPEC benchmark programs. RSFT-Detect handles system calls differently depending on which category they belong to.

- *Synchronize*: This category includes system calls that have no predictable program state after system call execution. These system calls cannot be executed redundantly (e.g. `sys_read` that writes data to memory). Some read operations, such as destructive read from standard input or other devices, can only be executed once. After the read, the content will be destroyed and not be available for a redundant read. RSFT-Detect can tell this kind of read from the file descriptor argument of `sys_read` call. RSFT-Detect must conduct `sys_read` system call only once, and copy the data to both program instances' private memory space to ensure identical program state in the subsequent execution. `sys_lseek` has the same semantics that need to be respected.

- *Asynchronize duplicate*: Some system calls, such as `sys_brk`, have no predictable return values, but its execution and return values must be duplicated in both program instances to ensure correct program execution. These system calls need to have identical return values in both program instances, but barrier synchronization is not necessary. RSFT-Detect handles this kind of system calls by allowing the first instance that invokes the call to execute the call and keeps a record of its return values. When the other instance invokes the same call, RSFT-Detect compares their arguments for faults detection, then enforces the call to do the same work in the second instance. In the example of `sys_times`, the recorded return value is returned to the second instance.

- *Asynchronize single*: This category includes system calls with return values that can be speculated with high confidence, but cannot be executed redundantly, such as `sys_write`. Among all system calls invoked dynamically on SPEC benchmark program, 72.59% calls belong to this category. RSFT-Detect does not require barrier synchronization for these system calls. If one copy of the process invokes `sys_write` ahead of the other process, it can continue executing the rest of the code without waiting for the system call to complete. When the other process reaches the point of issuing `sys_write`, it will compare the arguments of this system call and the memory content it is going to write for fault detection. If any fault occurs, RSFT-Detect is still able to report the fault to users.

3.5 Speculation

Previous work on transient fault detection, such as PLR proposed by Shye et al. [49], also explored the method of monitoring program execution at system call level to find mismatched arguments. However, these approaches conservatively require both processes (the original and the redundant) to synchronize at each system call. Some system calls can be executed redundantly, while some cannot. Both processes are allowed to continue execution only after non-redundant system call returns from the kernel, when its return value is duplicated and sent to both processes for further computation. This kind of barrier synchronization adds considerable overhead to program execution time. Evaluation in Chapter 6 discusses the performance in details.

RSFT-Detect, on the other hand, does not require barrier synchronization between processes when the results of the system call can be speculated. For example, in line C in Figure 2.3, `printf` invokes system call `sys_write`, which returns the number of bytes transmitted if it succeeds. RSFT-Detect always speculates the return value to be the number of bytes supposed to be transmitted according to the input parameter. The first program instance that invokes this `sys_write` call is allowed to continue execution without waiting for this system call to complete, as shown in Figure 2.4. This system call is only executed by the kernel when the other program instance also invokes the same call. This high-confidence value speculation allows one process to proceed without waiting for the other process or the system call to complete in kernel mode. A large set of the existing POSIX system calls such as `sys_write`, `sys_munmap` can be speculated in this way.

3.6 Misspeculation Detection and Recovery

Misspeculation occurs when the results of a system call differs from what RSFT-Detect predicts. This means the speculatively executing program instance must be discarded due to a wrong prediction made earlier. RSFT-Detect features a fast misspeculation detection and recovery scheme to restart program execution from the point of misspeculation.

In order to efficiently record program state, RSFT-Detect exploits system call `forks` copy-on-write semantics. If the tracing process detects misspeculation, it kills the speculative process and duplicate the non-speculative process for later computation. Copy-on-write semantics have previously been used to implement speculative systems [40]. Compared with transactional memory, this approach has less speculative overhead.

To inject a `fork` system call into the no-speculative process, without rewriting program's binary, RSFT-Detect follows the algorithm described in Algorithm 2. In the following algorithm, `W1` refers to the first invocation of the program binary, `W2` refers to its replicate, and `W` refers to either one of the worker processes (`W1` or `W2`).

Algorithm 2 Misspeculation Detection and Recovery

```
1:  repeat
2:    intercepts system call sys from a traced process W(either W1 or W2)
3:    // Code for fault detection
4:    if sys's return value was speculated in W2 then
5:      // symmetric if W1 is the speculative process
6:      if W1.Regs.rax != W2.specvalue[sys] then
7:        kill(W1.pid, SIGKILL)
8:        // duplicate the non-speculative process W2
9:        make a copy of the register file of W2
10:       W2.Regs.rax = SYS_fork
11:       // the instruction SYSCALL is 0F 05, which takes 2 bytes
12:       W2.Regs.rip = W2.Regs.rip-2
13:       continue executing the system call
14:       // program will re-enter interrupt and execute system call fork
15:       wait for kernel finishes executing SYS_fork
16:       W1.pid = return value
17:       copy back original register file of W2
18:       continue executing the program
19:     end if
20:   end if
21: until W1 and W2 both exit
```

For the code example in Figure 2.3, RSFT-Detect speculates the number of bytes being transmitted by system call `sys_write` and allows the process arriving first to proceed without waiting for the other process. When the other process calls `sys_write`, RSFT-Detect executes `sys_write` and then compares the return value with the speculated value. If the two values are the same, both processes proceed normally. Otherwise, misspeculation happens. The recovery process is demonstrated in Figure 3.3. When misspeculation is detected upon the return of the real system call (as shown in Figure 3.3), RSFT-Detect sends a signal to kill the misspeculated process. Another signal is sent to the program instance with correct values to force a `fork` system call. A new process is then spawned from `Program`, and continues the rest of program execution redundantly with the correct program state.

3.7 Virtual Memory Space Synchronization

To compare execution of two processes, the original one and its redundant copy, the fault detection may have to compare some virtual memory addresses at some system call. This is crucial to ensure identical memory layout of all processes to eliminate false-positives. As a result, some system calls, such as `sys_mmap`, must be executed in all copies of processes for a program to continue. Additionally, these two processes must have identical virtual address table to start with. In systems with address space randomization (ASR), independence processes have randomized virtual address mapping even for executing the same program binary.

One solution to this problem is to disable address space randomization entirely by configuring the operating system. However, this also disables the ability of other applications, for example security sensitive applications, from benefiting from ASR.

RSFT-Detect solves this problem by allowing the first process that reaches `sys_mmap` to execute first. Upon returning from the system call, RSFT-Detect keeps a record of the virtual memory address that this call allocates in this process. When the other the process invokes `sys_mmap`, RSFT-Detect first compares their arguments for transient fault detection, then forces this call to map to the same virtual address as the first process, by setting the MAP_FORCED flag. Because both processes maintain the same memory layout, both processes should be in the same program state before and after the system call.

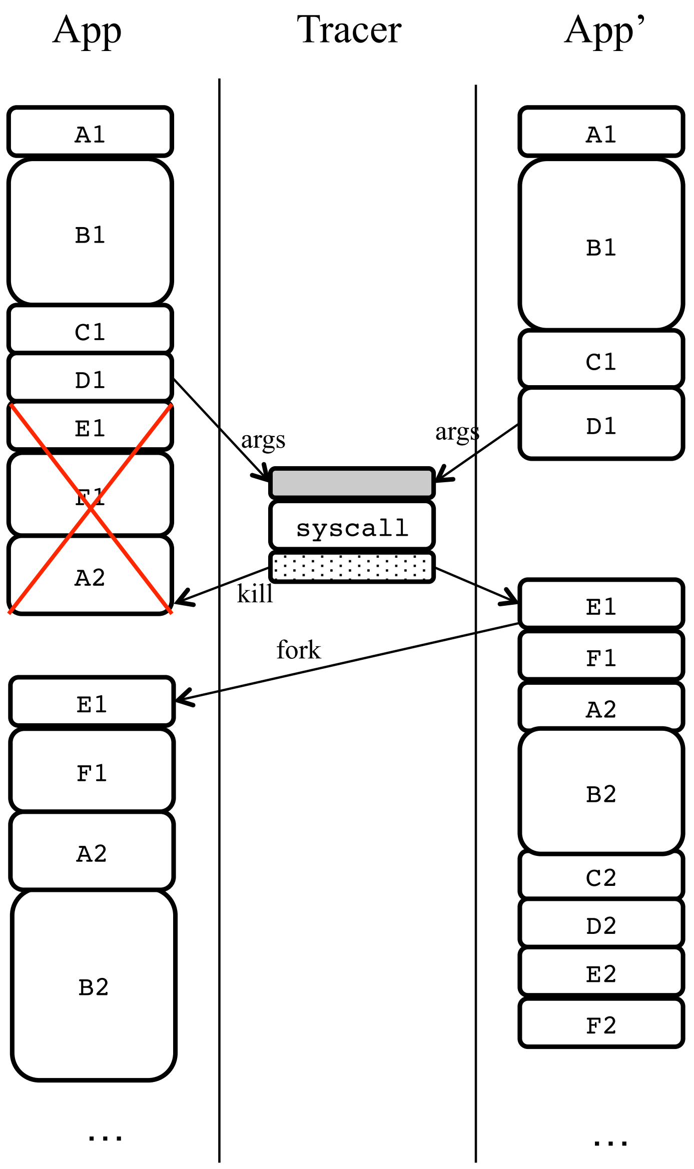


Figure 3.3: Misspeculation detection and recovery process in RSFT-Detect

There are also cases when `sys_mmap` intends to map files to a process' virtual memory space and performs read and write operations. By intercepting mmap system call and checking its arguments, RSFT-Detect can identify such requests and map that file in RSFT-Detect's own memory space. RSFT-Detect then returns a protected page address to the two program instances. When these programs intend to access the protected page, a signal is sent to and trapped in RSFT-Detect. At that point, RSFT-Detect checks for transient faults, then performs the actual read and write access only once on the memory mapped file.

3.8 Signal Handling

In RSFT-Detect, signals sent to the program and signals raised from one copy of the program must be handled in a way that is transparent to the user. Failing to handle these signals will result in either false alarms, or incorrect program behavior. All existing software-only transient fault tolerance techniques either ignore these signals or report them as transient faults despite of what they really are. This section discusses two different types of signals and the methods RSFT-Detect take to handle them.

Internal Signal: Signals can be raised from program itself, in the absence of transient faults, either as a program bug or designed the program behavior. However, some transient faults can cause a program to raise an unexpected internal signal. For example, a particle strike may flip a bit in a register that holds a memory address the program loads from. Note that the two identical program instances do not share memory space and have different set of register files. Consequently, a transient fault may cause a segmentation fault raised by one of the processes, but not both of them. Similarly, other exceptions, such as divide-by-zero, may also occur as a result of a transient fault. To distinguish a transient fault from a normal internal signal raised by the program, such as signals caused by program bugs, RSFT-Detect traps all signals raised by both processes. If the two processes both raise the same internal signal, it is an expected signal and must be handles in the default way. Otherwise, RSFT-Detect alerts the users the existence of a transient fault.

External Signal: As a transparent transient fault detection technique, RSFT-Detect maintains the original deterministic behavior of the original program. External signals may cause non-determinism among the two copies of programs. For example, the user may press `ctrl-c` from command line, which sends a SIGINT signal to the program. RSFT-Detect should make sure the two program copies behave as if only one program is running and abort the program.

To achieve this, RSFT-Detect registers special signal handlers for all external signals. Specialized signal handlers are registered at the beginning of the program. When an external signal is received, the corresponding signal handler is called and proper actions are taken. For example, in the case of a SIGINT sending to the program from command line, RSFT-Detect communicates SIGINT to both processes, terminates their execution and kills itself as well.

There are cases where one fault process may bypass the signal raising code. RSFT features a timeout watchdog process, which will raise timeout signal if the program has not made progress after a pre-defined threshold. RSFT will handle this case as a transient fault. This threshold is typically application specific. In the evalution of this dissertation, the threshold was set to ten times of a typical run's execution time.

3.9 Optional Memory Page Walking

To protect program execution from transient faults in memory sub-systems, RSFT-Detect adopted process-level duplication and use redundant memory space. However, two processes, one forked from another, share the same physical memory page, if no value is written to that page. As a result, memory faults occurring in pages that are shared by the two processes can be read by both program copies hence introducing faulty values. This means the two copies will produce the same faulty results, which may cause the system to fail severely.

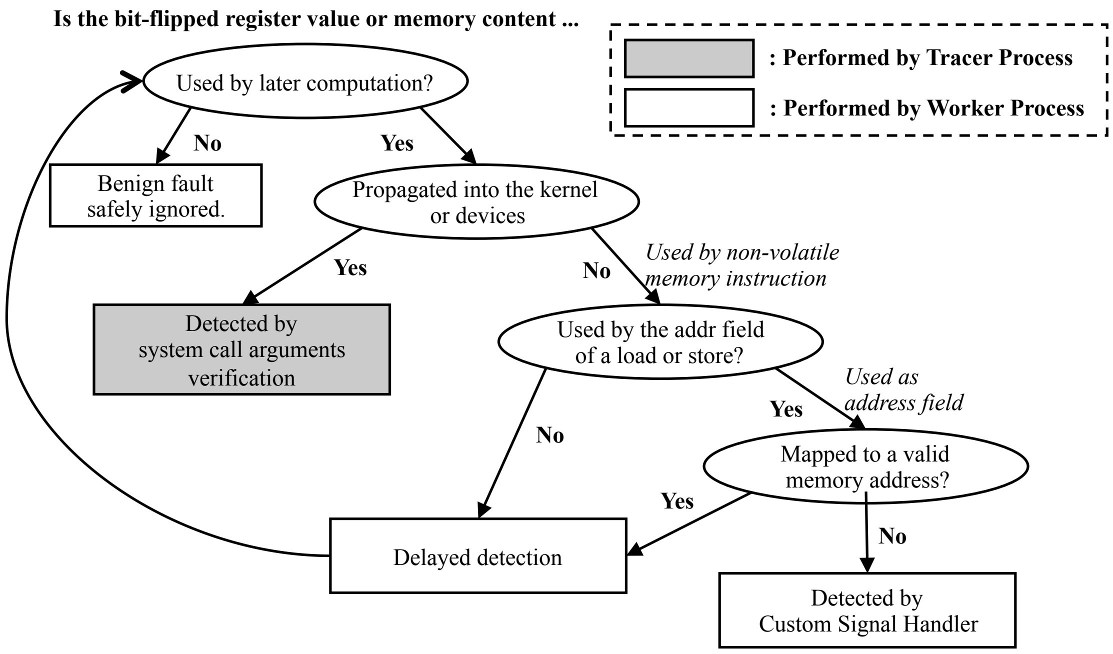
RSFT-Detect solves this problem, by optionally turning on a knob to perform automatic memory page walking. The idea is to check the two processes' physical page table periodically. If any of the pages that are physically in memory has the same physical page address for both processes, RSFT-Detect automatically loads a word from the page, and immediately writes the same value back, utilizing the copy-on-write feature of forked processes. The operating system will automatically create another physical page as a tainted copy.

There is a trade-off between memory consumption and memory fault coverage. By enabling memory page walking, there is a better chance for RSFT-Detect to detect memory transient faults, at the cost of using more physical memory during redundant execution. Disabling this memory page walking will reduce the size of active physical memory usage, but results in larger window of vulnerability in practice.

3.10 Transient Fault Detection

A combination of the above techniques gives RSFT-Detect the ability to detect a wide range of transient faults in both processor and memory subsystems. Figure 3.10 lists several results of transient faults, and how RSFT-Detect handles each of them.

- **Register Value**. A transient fault may alter a register value that never flows into the following computation of the program. These faults can be safely ignorely. If the altered register value directly changes the values that escapes the SoR, these faults can be detected via value comparion in RSFT fault detection. If a fault changes an address of a memory access operation, the program may trigger a segmentation fault. If a segamentation fault is found in only one of the work processes, this asymmetric


Is the bit-flipped register value or memory content ...
: Performed by Tracer Process
: Performed by Worker Process
Used by later computation?
No
Yes
Benign fault safely ignored.
Propagated into the kernel or devices
Yes
No
Used by non-volatile memory instruction
Detected by system call arguments verification
Used by the addr field of a load or store?
Used as address field
Yes
No
Mapped to a valid memory address?
Yes
No
Delayed detection
Detected by Custom Signal Handler

behavior is identified as a transient fault. Otherwise, transient fault detection will not catch this fault when it occurs. Instead, this fault will be caught in later fault detection via value comparison.

- **Memory address**. If a memory content is modified by a transient fault, this value can be either instruction or data. In case the value is an instruction, and will be executed, the two worker processes have high probability of executing different binary and end up sending different values out of SoR. This kind of faults is captured by RSFT. Otherwise, this altered instruction can be safely ignored. If the memory content holds a value to be loaded to the program, RSFT detects such fault like register values discussed above. Otherwise, this fault will not change the program behavior, hence not the target of transient fault detection.

3.11 Window of Vulnerability

A fault occurring in RSFT-Detect itself may cause an unrecoverable error or erroneous results or undefined program behavior. Additionally, RSFT-Detect does not protect the operating system and its services executed in kernel. Transient faults occurring in the kernel code may still cause program failure.

Although RSFT-Detect itself represents a single point of failure, it only occupies the CPU for a very short duration and only consumes a few clock cycles. Typically the RSFT-Detect process only takes less than 0.01% CPU time throughout program execution. The probability of transient faults occurring registers while RSFT-Detect code is running is extremely low.

The memory consumption of RSFT-Detect is 9.5MB peak. If a transient fault hits the main memory that holds data used in the tracing process, it may lead to program crash or wrong output. However, such faults may still be benign faults or be detected via transient fault detection. For example, if the fault flips a bit of a word in memory, which is over-written before the value is loaded, the fault is a benign fault. If a memory fault changes the value of a word in memory which results in a value mismatch in fault detection code, it can also be detected.

Chapter 6 evaluates RSFT-Detect's window of vulnerability by simulating transient faults in both register files and main memory.

Chapter 4

RSFT Transient Fault Recovery

The previous chapters presented transient faults detection at program runtime using program instance duplication and runtime checking. However, detecting a transient fault is only the first step of fault tolerance. Detecting a fault only prevents program from corrupting outside visible program state and enforces data integrity. Detection alone does not guarantee un-interrupted correct program execution in the presence of transient faults. In an environment where transient faults occur frequently, users may never have a single run without transient faults corrupting critical values. A system must be able to recovery from a fault to be truly fault resilient.

This chapter describes the transient fault recovery system of RSFT, called RSFT-Recover . RSFT-Recover is a novel program execution checkpointing and recovery system that performs cheap and fast program checkpointing, program state verification and fast program state recovery.

This chapter first introduces the background of transient fault recovery. Section 4.2 discusses several common fault recovery mechanisms proposed previously and their limitations to motivate the RSFT-Recover. Section 4.3 illustrates the overall design of RSFT-Recover . The following sections introduces the implementation details of RSFT-Recover .

Section 4.5 discusses RSFT-Recover's window of vulnerability and compare it with existing fault recovery proposals.

4.1 Transient Fault Recovery

After a transient fault is detected, the program may behave differently depending on the result of the transient faults.

- Benign Faults. These transient faults alter values held in registers or memory, but does not propogate to the output or devices. The program can continue executing and produce correct output after a fault has occurred.
- Non-benign faults. Program execution hit by a transient fault may produce erroneous values to the outside world. In such cases, the program either cannot complete execution, or generates wrong output. When these faults are detected, a transient fault recovery is necessary for the program to finish correctly.

RSFT operates as a layer between applications and the underlying operating system. As a result, transient faults detected by RSFT are always non-benign faults. To complete program execution correctly and fully tolerant the transient faults, effective fault recovery is necessary for programs protected by RSFT.

Among the previous related work discussed in Chapter 2, only a small number of transient fault tolerance techniques cover transient fault recovery after a fault is detected. The following section discusses several representative transient fault recovery solutions, their advantages and disadvantages. The transient fault recovery scheme is RSFT is introduced in later sections.

Previous transient fault detection techniques, such as SRT [29], requires frequent checkpointing to recover from faulty states to safe states.

4.2 Existing Techniques and Limitations

4.2.1 Program Re-execution

The program execution is stopped after detecting fault, and is restarted from the beginning, hoping to have a fault-free execution in the second run. This approach has several problems itself.

- The program may have already resulted in outside-visible effects, such as sending network packages. Re-executing from the very beginning of the program results in recomputing correct results computed previous to the fault, hence more resource consumption.
- The probability of having a transient fault in the second run is independent of the previous run. Hence it is also possible to have a soft error in the second run as well. Continuously stopping and re-executing a program, especially for long-run scientific programs that may take up to weeks or months, is time-consuming and not efficient.

4.2.2 K-Modular Redundancy

Some existing techniques uses or triple-modular redundancy(TMR) in either hardware, such as Boeing airplanes [65] or software[42], PLR [49] to recovery from a transient fault. TMR means that either three identical program/hardware copies, or one main code and two redundant copies are computed simultaneously. This allows the fault to corrupt one execution at a time, as long as the other two can agree on the critical values. By using this scheme, any single-even-upset can be corrected as two of the versions still have the correct data. After a single-event fault is detected, the corrupted values are ignored and corrected. If any independent memory image is corrupted, the particular program instance is also discarded.

As a natural extension of triple-modular redundancy, some fault recovery techniques employ *k*-modular redundancy and majority voting to correct program execution from the failure point one, in the presence of multiple transient faults. These recovery techniques typically use more than three identical execution contexts, and compare the values that escape the SoR to detect transient faults. If any of the values mismatch with the majority of other copies, the value is considered corrupted and a transient fault is detected.

To recover from the fault, the identical majority values are considered correct and used in later computation. The corrupted value is thrown away and that execution context is also discarded. A system with majority-voting fault recovery can survive *k* faults is called *K-fault tolerant*. Such a system requires at least *k+1* processes to tolerance *k* faults. In situations where faults occur more than once during program execution, the cost of majority voting recovery can be very high both in term of performance and power consumption.

Some techniques can spawn another execution context with correct values from the failure point on, to maintain *k* active program instances at any time. This solution may be effective under single-event-upset model. However, if transient faults occur more frequently than expected, it is possible several active copies contain different corrupted values at the same time. Majority voting may not get a majority vote. Moreover, some program instances may present identical values like other copies, but may contain other corrupted values in memory. Duplicating another instance from these instances may lead to future program failure.

In addition, using more than three processes for fault recovery consumes more computation resources, such as CPU time and memory, than using two processes. Power consumption is also increased with the increase of execution contexts. This is an effective yet not ideal way to recovery from transient faults.

4.2.3 Checkpointing

Other techniques [14, 38, 39, 52] keeps track of correct program states via checkpointing program execution. These checkpoints can be used later if any transient fault is detected and program needs to roll back to a previous correct version.

In practice, the program, or another checkpointing process, creates checkpoints periodically, either in memory or some external storage device. When a fault is detected, the program rolls back and resumes execution from the last program checkpoint. This approach does not have the problem of keep hitting a fault during execution. However, it still suffers from the problem that some critical values may have escaped SoR between checkpoint and the point a fault is detected. One solution is to checkpoint more frequently, ideally before or after every instruction that may cause outside-visible behavior. However, frequent checkpointing slows down program significantly. The evaluation demonstrated in Chapter 6 will discuss this problem further.

To solve this problem, specialized hardware modules were proposed to perform checkpointing to maintain correct hardware state. Additional IO operations bookkeeping are necessary for fault recovery using checkpointing.

In addition, software runtime checkpointing typically introduces significant performance overhead. For example, some existing checkpointing algorithms stalls program execution, make a record of all register files and memory state, keeps the record in either memory or disk files, then resume program execution.

4.3 RSFT-Recover

This dissertation proposes a runtime lightweight checkpointing and fault recovery technique for transient fault recovery. This technique provides automatic program checkpointing, outside-visible behavior bookkeeping, and program re-execution from last verified checkpoint.

Compared with previous proposed fault recovery techniques discussed above, RSFT-Recover does not need to re-execute the whole program from the beginning, hence reducing the probability of transient fault occurring during program execution.

4.3.1 Structural View

Like the fault detection discussed in the previous chapter 3, RSFT-Recover automatically duplicates the program binary execution into two copies. Current security policies of modern operating systems only allow a process' parent(s) to trace and intercept system calls. Therefore, slightly different from the implementation of fault detection only, RSFT-Recover first creates a process for conducting the process (T) tracing and fault detection. Process T will create a process (W1) to invoke the program binary and duplicate W1 in another process W2 for redundant execution. Figure 4.1 illustrates the overall structure of fault recovery.

4.3.2 Runtime non-invasive checkpointing

To restart a program from a correct previous version after a fault is detected, the program needs to create checkpoints from time to time. The runtime checkpointing of RSFT-Recover is implemented via duplicating a process' state, including register files and memory content. The duplication process utilize the `fork` system call provided by operating systems. `fork` system call is designed to duplicate a process, with all its memory pages marked copy-on-write(COW). The two identical program instances, namely W1 and W2, each create a checkpoint using `fork` system call independently, but at the exact same program point.

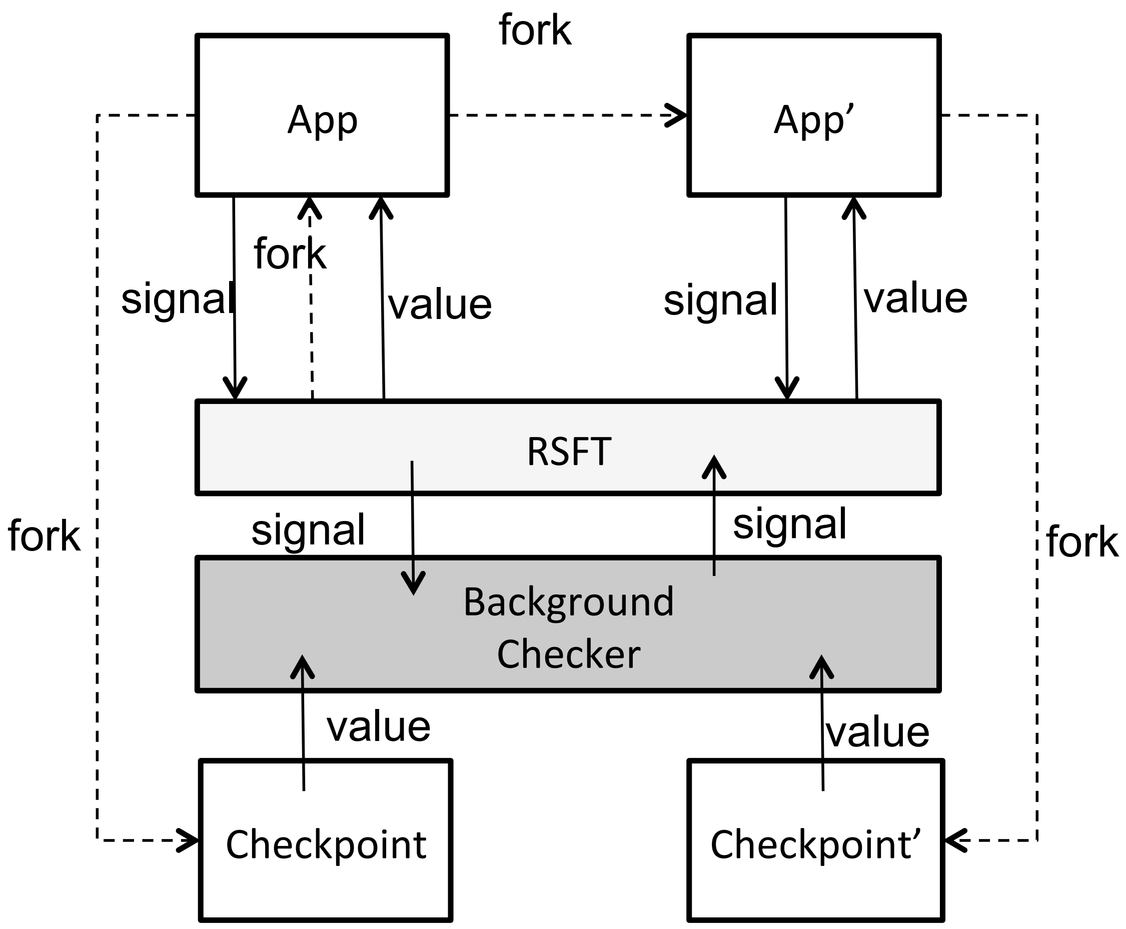


Figure 4.1: RSFT-Recover Structure Overview. Dashed line indicates forking a child process then transfers tracing control to other processes.

The program point RSFT-Recover choose to create a checkpoint is predefined as an environmental variable in term of number of system calls (NSYS) in the program. RSFT-Recover checks the value of the variable at program runtime and create checkpoints every NSYS system calls.

To invoke a `fork` system call in W1 and W2, RSFT-Recover need to either modify the binary of W1 and W2, or inject a system call in a non-invasive way. In the implementation proposed in this dissertation, RSFT-Recover inject a `fork` system call using Algorithm 3.

4.3.3 Background Process Image Verification

After a checkpoint process is created, it is trapped by the tracer process (T). T communicates its PID to the background scrubbing process through software communication queue described in [19]. The scrubbing process consumes the PIDs and performs background memory content and register value verification. Algorithm 4 describes the process of memory scrubbing after a checkpoint is created.

4.3.4 Program State Bookkeeping

One common problem of previous recovery techniques is that re-executing the program, either from the beginning or from a checkpoint, involves re-issuing some instructions that has side-effects. For example, a program may send signals to another process, or send a package over the network to another system, or transfer values to hardware devices, after a verified program checkpoint. Most of these cases are non-reversible. After a program recovers from a checkpoint, if the program executes these side-effecting instructions again, the program may behave incorrectly from the outside world's view.

To solve this problem, RSFT-Recover keeps track of all program behavior that may be visible outside of its SoR. The nature that RSFT operates at the system calls granularity

Algorithm 3 Automatic Runtime Checkpointing

```
1:  repeat
2:      intercepts system call sys from a traced process W(either W1 or W2)
3:      nsys++
4:      if nsys == NSYS then
5:          // create a checkpoint
6:          if process W is entering a system call then
7:              make a copy of the register file of W
8:              W.Regs.orig_rax = SYS_fork
9:              continue executing the system call
10:             wait for W to return from SYS_fork
11:             NewPID = W.Regs.rax
12:             enqueue(NewPID)
13:             copy back original register file of W
14:             W.Regs.rax = sys
15:             // the instruction SYSCALL is 0F 05, which takes 2 bytes
16:             rip = rip-2
17:             continue executing W
18:             // program will re-enter interrupt and execute the original system call sys
19:         else
20:             make a copy of the register file of W
21:             Regs.rax = SYS_fork
22:             // the instruction SYSCALL is 0F 05, which takes 2 bytes
23:             rip = rip-2
24:             continue executing the system call in kernel
25:             // program will re-enter interrupt and execute system call fork
26:             wait until the system call returns from SYS_fork
27:             NewPID = W.Regs.rax // return value
28:             enqueue(NewPID)
29:             copy back original register file of W
30:             continue executing W
31:         end if
32:     else
33:         perform fault detection
34:     end if
35: until W1 and W2 exit
```

Algorithm 4 Background Memory Scrubbing

```
1:  repeat
2:     newPID1 = dequeue();
3:     newPID2 = dequeue();
4:     openMemoryMapFile(newPID1, newPID2
5:     openMemoryFile(newPID1, newPID2)
6:     for all memory page P in memory of newPID1 and newPID2 do
7:        touchMemoryPages(P)
8:        read page P from memory of newPID1, newPID2
9:        compare memory content from two versions
10:       if memory contents mismatch then
11:          discard this checkpoint
12:       else
13:          keep this checkpoint
14:          discard the previous checkpoint
15:       end if
16:    end for
17: until signaled to exit
```

allows RSFT -recovery to bookkeep the program states in term of system calls. After a program checkpoint is made, RSFT-Recover keeps track of the number of system calls issued from each program instance. Note that RSFT compares the arguments of system calls at each call site, the difference of number of system calls issues from the two program instances are less or equal to one. Once a transient fault occurrence is confirmed, RSFT-Recover tries to recover program from the last checkpoint. After the program restarts from the last verified checkpoint, RSFT-Recover is able to replay the program execution without re-issuing instructions that cause outside-visible behaviors.

4.4 Runtime Fault Recovery

When a transient fault is detected during program execution, RSFT-Recover picks up a previous correct checkpoint and resume program execution from the checkpoint on.

4.4.1 Checkpoint Process Resuming

The checkpoint processes are duplicates of the original working processes W1 and W2. They are stopped by the scrubbing process for memory content comparison. After the process image comparison is successful, they are still stopped at exactly the same program point and not allowed to continue execution.

Program execution recovery is two simple steps, as illustrated in Figure 4.2:

1. Kill the existing two working processes.

2. Signal the last verified checkpoint processes. The checkpoint processes are woken up by signals to continue executing the rest of the program.

4.4.2 System Call Replay

After the program resumes from a previous correct checkpoint image, RSFT-Recover keeps track of the number of critical system calls such as `read, write`. This number is compared with the number of correctly executed system calls after the checkpoint. If the number is less than the previously successfully committed system call number, this system call is bypassed. RSFT does not allow this system call to flow into the operating system or causing any outside visible behavior.

Note that some system calls, such as `mmap, mremap`, are still performed to ensure future correct program execution. Only those that will send values to outside world will be emulated and not replayed.

However, there are some cases that this replay may violate program semantics. For example, if an application is supposed to send a heartbeat network message at a certain frequency, skipping such system calls may change how the outside world sees this application. While this dissertation does not handle this case, it is possible to either notify users

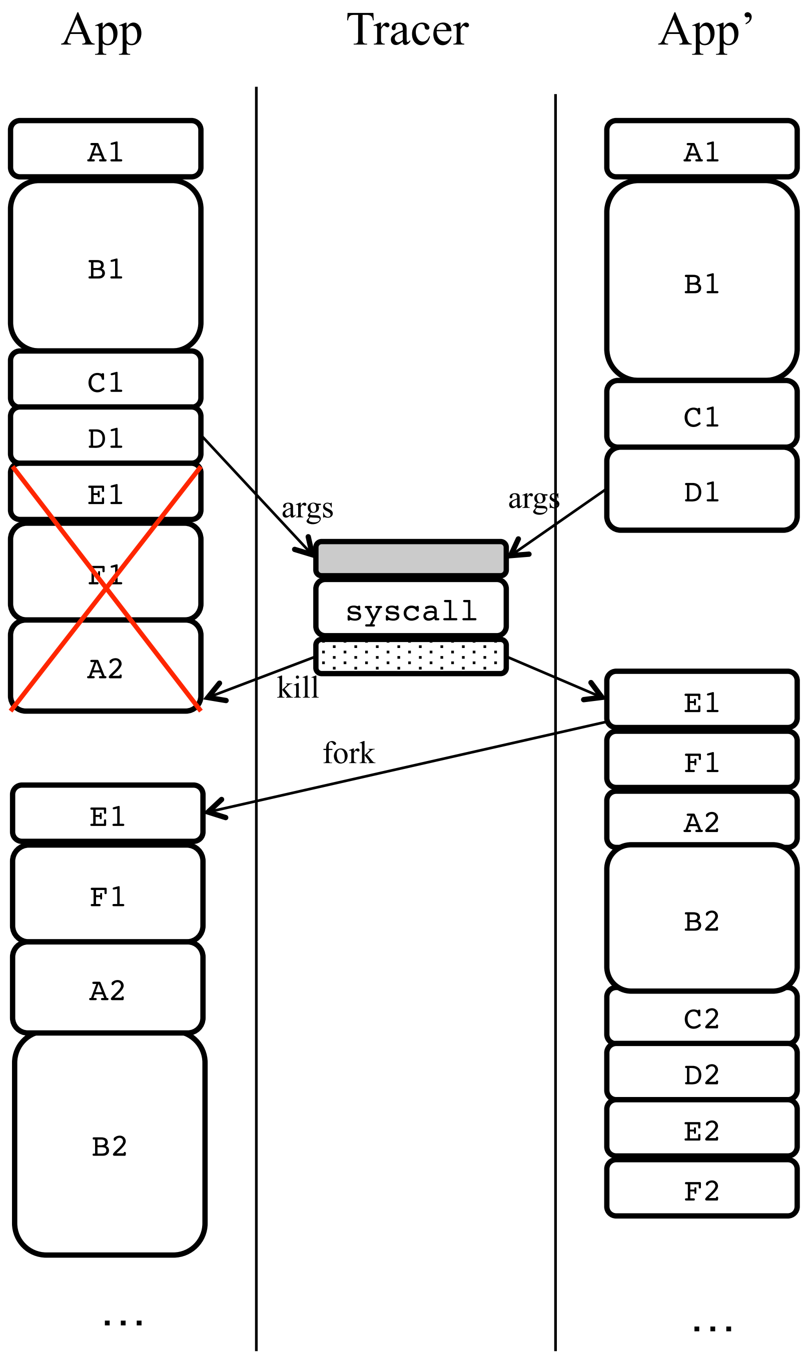


Figure 4.2: Transient fault recovery timeline in RSFT

to ignore temporary out-of-sync nodes or modify the receiver of such network messages to be aware of the temporary service outrage.

4.5 Window of Vulnerability

RSFT-Recover has a small window of vulnerability. First of all, the recovery process itself is a single point of failure. After a transient fault is detected, RSFT-Recover wakes up the previous checkpoint and tries to restart program execution from the last correct program state. However, if another transient fault occurs during the recovering process, the program may (1) fail to restart, or (2) restart with a faulty program state. In the former case, the timeout mechanism will ask the program to restart from the very beginning. Although this is a slower than usual recovery, it still manage to recover program execution transparent to the user. In the latter case, the faulty program instances may either result in RSFT detecting the fault later via value verification, or incorrect program output.

Given the potential occurrence rate of transient faults in common environments, it is extremely unlikely that the recovering process is hit while performing the recovery. But in severe environments with high radiation rate, this is a window of vulnerability for RSFT protected programs.

Another vulnerable window of RSFT-Recover is the verification process. During memory verification, if the memory content of the checkpoint processes, or the verification result (match or mismatch) is corrupted at runtime due to a transient fault, RSFT-Recover may incorrectly conclude that one of the working processes is affected by a transient fault. However, RSFT-Recover will not raise a false positive alarm of transient fault in such a case. Instead, RSFT-Recover will discard this version of checkpoint and hoping to get a correct checkpoint next time.

RSFT-Recover also registers signal handlers for the background process. If the background verification process itself suffers from a transient fault, which results in a process crash, RSFT-Recover will trap the signal, and create a new process to continue perform scrubbing from the failure point on. However, all previous checkpoint information will be lost.

If any memory content of the checkpoint is corrupted before it is used for recovery, it may lead to future program failure. However, because these two checkpoint processes are hanging there, waiting to be waken up. The operating system will schedule them off the processors and memory system most of the time, the window of being hit by a transient fault is only during process image comparison, which is extremely low compared with the other processes or the whole program execution.

Note that RSFT-Recover does not introduce false positives to transient fault detection, since it is independent from transient fault detection RSFT-Detect. RSFT-Detect can work with other recovery mechanism as well in a plug-and-play fashion.

Chapter 5

Evaluation Methodology

This chapter describes the methodology used to evaluate the reliability of RSFT presented in this dissertation. Section 5.1 discusses the standard transient fault model, and a more realistic fault model used for evaluation in this dissertation. Section 5.2 explains the metrics used to measure reliability of RSFT. Section 5.3 details the simulation details and evaluation method used to measure the metrics.

Section 5.4 describes the evaluation environment in this dissertation. And Section 5.5 presents an analysis of the benchmark characteristics. Section 5.6 inroduces the methods used in evaluation on a set of 23 for reliability, performance, and resource consumption.

5.1 Transient Fault Model

In recent fault tolerance research, *Single Event Upset(SEU)* model is commonly adopted to evaluate the reliability of techniques. This model assumes that one and only one transient fault occurs during one program execution, and this fault flips exactly one bit. The SEU itself is not considered permanently damaging to the transistor's functionality.

Although SEU model is commonly adopted, it is also beneficial to study the reliability of fault tolerance under the *Multiple Event Upset (MEU)* model. The MEU model means that there can be more than fault occurring per program execution. These faults may flip exactly one bit, or flip multiple bits at a time. In reality, radiation may flip a bit and one or more bits that are physically close to the flipped bit. Compared with the SEU model, MEU is more realistic and is useful to study the fault tolerance techniques when fault frequency is higher than expected. ECC memory can protect memory content against transient faults, but limited to SEU only. In cases where multiple bit are flipped within an ECC protection unit (e.g. a word), ECC is more likely to fail.

To compare with existing technique, we did experiments with MEU model. For each fault model, the simulation was done 3,000 times on each benchmark program. The evaluation is implemented using a processing tracing tool (`ptrace` on Linux) to simulate transient faults at runtime in both register files and memory space, in a way similar to [20, 50, 21]). Two bits of the same word are flipped in the program's register files or physical memory space.

5.2 Reliability Metrics

Transient faults may have different effects on program execution. Mukherjee et al. classify transient faults by these effects into the following categories [31]:

- **Benign Fault.** A transient fault that does not affect the outcome of the program is a benign fault. Wang et al. report that 85% of total transient faults are benign, and result in no externally visible program errors [61].

- **Silent Data Corruption.** If a fault induces the system to generate erroneous program outputs, it is said to have suffered silent data corruption. Silent data corruption, or SDC, is what all fault detection techniques are designed to detect and what fault recovery mechanisms are designed to recover from.

- **Detected Unrecoverable Error.** A detection-only protection mechanism does not recover the program from a fault, but prevents SDC through fail-stop mechanism, thereby avoiding any data corruption. An error of this kind is called Detected Unrecoverable Error, or DUE.

The target of this dissertation is to detect and recover from the faults that cause SDCs. RSFT benefits from the insight that minimizing the cost of detecting benign faults can improve program performance by a large amount.

The average Mean-Time-To-Failure (MTTF) time is a widely accepted metrics to evaluate the reliability of a fault tolerance technique. At the time of a fault injection, a timestamp is recorded. If the fault does not lead to any failure or detected fault, this timestamp is discarded. Otherwise, another timestamp is recorded at the time of fault detection or program failure. The difference between these two timestamps is called Mean-Time-To-Failure (MTTF). The distribution of MTTF out of 3000 runs of each benchmark program is demonstrated in the evaluation section.

Mean Time to Failure does not describe the reliability of a fault tolerance technique entirely. If one technique enables the program to execute twice as faster as the program protected by another technique, the second program, since taking longer to run, is twice more likely to be hit by transient faults. Some previous work used other metrics, such as Mean Instruction To Failure (MITF) and Mean Work To Failure (MWTF) [44]. These methods are useful when all instruction are equally important and takes roughly the same time to execute, or when the definition of work is the same across all techniques in comparison.

This dissertation adopts the metrics of MTTF. Typically, the faster a program executes, the less likely it execution is affected by transient faults, which is correlated with MTTF.

Reason	Symptom	Modelling
Instruction	instruction corruption	Modify the value in a random memory address that maps to the program binary
Physical Memory	instruction corruption or data corruption	Modify random memory address that may contain instruction or data
Memory Bus	instruction fetch error or data corruption	Modify random memory address that may contain instruction or data
Register File	data corruption	Flip bits in a random general-purpose, XMM, floating point, or register
Program Counter	incorrect program flow	Flip bits in the PC register
Control Logic	PC value corruption	Bit flip in PC register
Re-order buffer	data corruption	Bit flip in a random register or memory content
load store queue	data corruption	Bit flip in a random register or memory content

Table 5.1: Transient fault types and modeling

Given that the goal is to protect program execution from transient faults, it is reasonable to simply measure MTTF as the simulation is injection a constant number of faults per run, not per time unit.

5.3 Transient Fault Simulation

This dissertation considers transient faults in register files, physical memory, and control-logic. Table 5.1 shows various types of transient faults and the way they are modelled in this dissertation.

All reliability evaluation demonstrated in this dissertation injects faults into *all three processes*(two worker and one tracer). This fault injection method does not inject faults into the operating system itself. However, the time spent in the tracer process includes the time that all system calls spent in the operating system. By injecting faults into the tracer process and change its register or memory values, some faults in the operating system are also simulated.

5.3.1 Register Fault Simulation

Many prior work uses PIN dynamic instrumentation to inject faults into register files. This thesis uses another system-level approach. First, a profile run of the original program binary is timed to estimate how long it may take to execute the program. Before fault injection, our tool randomly selects one point in time, one random program instance, one random bit of a register, as well as one random register among general-purpose, floating point, XMM, and flag registers. During program runtime, this fault injection simulation issues an alarm after a random period of time. It then sends a signal to the randomly selected process, stops its execution, and flips the random bit of the selected register. The particular program instance then continues execution. Finally, the execution result of the fault-injected program is compared against the reference output to ensure that the RSFT-protected program's externally visible behavior is correct.

The drawback of this kind of simulation is that it only simulates architectural states including register files, instruction decoding, and program counter. Some work also simulates transient faults at a microarchitectural level [43], and an RTL level [61]. Because transient faults naturally occurs at the hardware logic level, simulation at a lower level, such as RTL level, has better accuracy. For example, the hardware simulation can inject faults into the bypass network, which is not visible to the architectural view. However, this approach is usually prohibitively slow for sizable programs such as SPEC benchmark suites. Mukherjee et al. proposed methodologies to estimate the architectural vulnerability factors(AVF) at microarchitecture level by sampling the AVF of a given structure in a given time [31]. These techniques can only simulate and cover a fraction of the program.

The reliability evaluation in this dissertation is pessimistic. For example, some benign faults, such as the faults that hit the processor but are automatically correctly before the value is loaded, or is microarchitecturally masked, are not simulated and represented in this dissertation. However, since these are benign faults and does not change program behavior, RSFT does not attempt to detect or recovery from them.

5.3.2 Memory Fault Simulation

Previous work only simulates transient faults in registers. The instruction-level redundancy and redundant multi-threading approaches only maintain one memory state, and rely on ECC memory to protect programs against memory transient faults. Process-based redundancy techniques maintain multiple memory states and can provide transient fault protection for memory. However, prior work using process-based redundancy, including PLR, did not do any experiments regarding memory transient faults. This thesis is the first to simulate and evaluate the detection of transient faults in memory for software transient fault tolerance techniques.

Similar to injecting faults into register files, memory fault injection involves a profile run, and a random selection of a program point. Subsequently, memory fault injection randomly selects a virtual memory address owned by one process, and randomly flips a bit of the value stored in that memory address. This memory address may contain data or text of the program. Memory faults are injected into all running processes (the monitoring process, and two program instances), with the likelihood of the occurrence of the faults being proportional to the amount of memory used by each process. Each program was executed 3000 times with one memory fault injected each time.

Memory faults are only injected into the physical memory. Although a process may use a large amount of virtual memory, the number of physical memory used is typically much smaller than the virtual memory pages. One reason is that virtual memory space includes not only stacks and heap, but also binary file and dynamically loaded libraries. Not all the pages are accesses during program execution. Another reason is that only recently used pages are swapped into the main memory for locality.

Because we are simulating real transient faults which occur in hardware, memory content that is not physically in memory is not supposed to be changed by any transient fault.

As a result, like the register fault simulation, this dissertation also picks random point in program execution, and halts the process execution at that point. After the process is paused, the injection process walks through the physical page table of the randomly selected process, make a record of pages that are currently in memory, and randomly pick a bit in a random memory address that belongs to those pages.

To demonstrate the advantage of memory fault tolerance using RSFT, the evaluation in this dissertation flips two bits of the same word stored in that random memory address. The reason is that ECC memory provides fault tolerance against transient faults. Guaranteed to detect and recover from single-bit flip faults, ECC memory may fail when multiple bits of the same protection unit are flipped at the same time. However, when a particle strikes and flips a bit, it is possible that the same particle beam may flip other bits that are physically near the already altered bit, which makes ECC memory vulnerable. RSFT compares the values instead of setting parity values of bits, therefore can detect multiple-bit-flip events.

Memory faults injected into RSFT itself may also be detected. For example, if a fault changes the value in memory that stores the system calls' arguments for later comparison, this fault will be detected when the values are compared against values from the other process later. Another example is that the memory fault injected may be transitively passed on to one of the processes and results in a value mismatch in later transient fault detection. However, if an injected memory fault changed a value in memory that is transitively passed on to both processes, and that value affects the final output, RSFT will not be able to detect it.

The disadvantage of this memory faults injection method is that the simulation is not at hardware level. Previous work on simulation memory faults, such as the ECC technique, evaluates its reliability by simulating transient faults at the hardware level. Hardware level simulation is more accurate than software level simulation. However, the simulation methodology used in this dissertation can approximate hardware simulation as close as soft-

Processor	Intel Core 2 Duo®Q6600
Processor Speed	2.4GHz
L2 Cache size	4096KB
RAM	8GB
Operating System	Linux 2.6.38
Compiler	gcc 4.6

Table 5.2: Platform details

ware simulation can get. Because RSFT is an application-level fault tolerance technique, memory faults in the memory space used by the unprotected programs are not simulated or evaluated.

This chapter presents an thorough evaluation on a set of 23 for reliability, performance, and resource consumption. The methodology used to evaluate RSFT is described in this chapter.

5.4 Experiment Setup

The program binaries for all benchmark programs are generated using compilation option -O2 with gcc. The performance of the original binaries are used as the baseline throughout the evaluation. Both performance and reliability evaluation were conducted by executing the binaries with RSFT on a 2.4Ghz Intel Core 2 Quad-Core (Q6600) machine running Linux 2.6.38.

5.5 Benchmark Programs

The benchmark program used for evaluation in this dissertation are from SPEC CPU 2000 and SPEC CPU 2006 benchmark suites. These benchmarks were selected to be included in SPEC benchmark suites as representatives of real-world applications, including general-purpose applications and scientific applications. For example, 433.milc is developed by

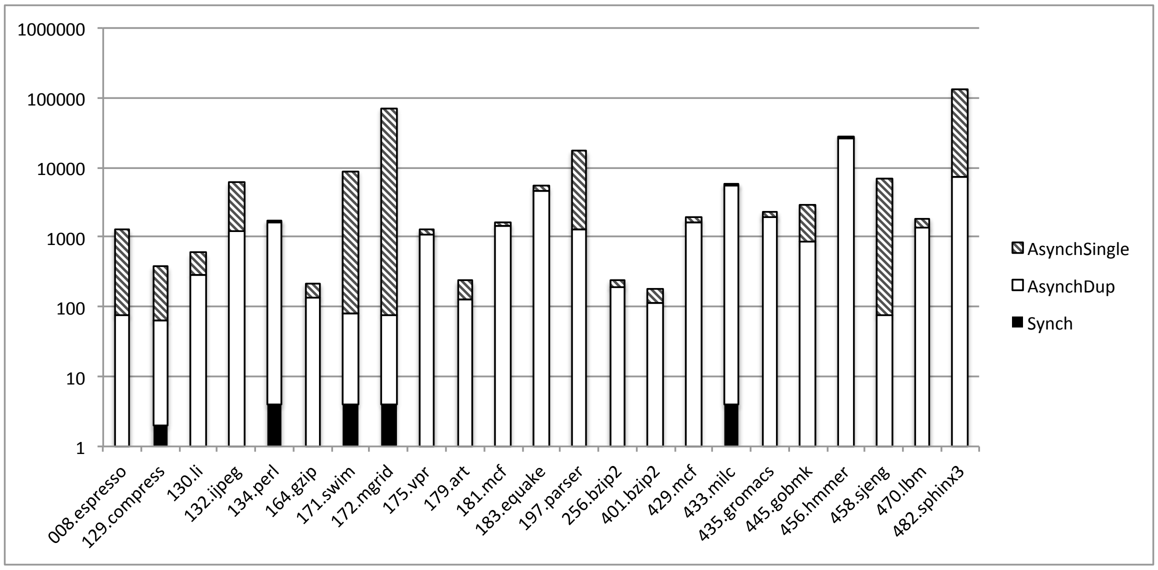


Figure 5.1: Number of system calls in each benchmark (log scale)

the MIMD Lattice Computation (MILC) collaboration for doing simulations of four dimensional SU(3) lattice gauge theory on MIMD parallel machines. This code is used for millions of node hours at DOE and NSF supercomputer centers. The programs selected for evaluation in this dissertation were chosen based on availability.

Figure 5.1 shows the number of system calls issued from each unprotected original benchmark program. Due to the huge variation of number of system calls (from several hundreds to a couple of hundred of thousands), the number of system calls are shown in the log scale of 10. Among all 298 system calls defined, about 45 of them are actually issued in the SPEC benchmark suite.

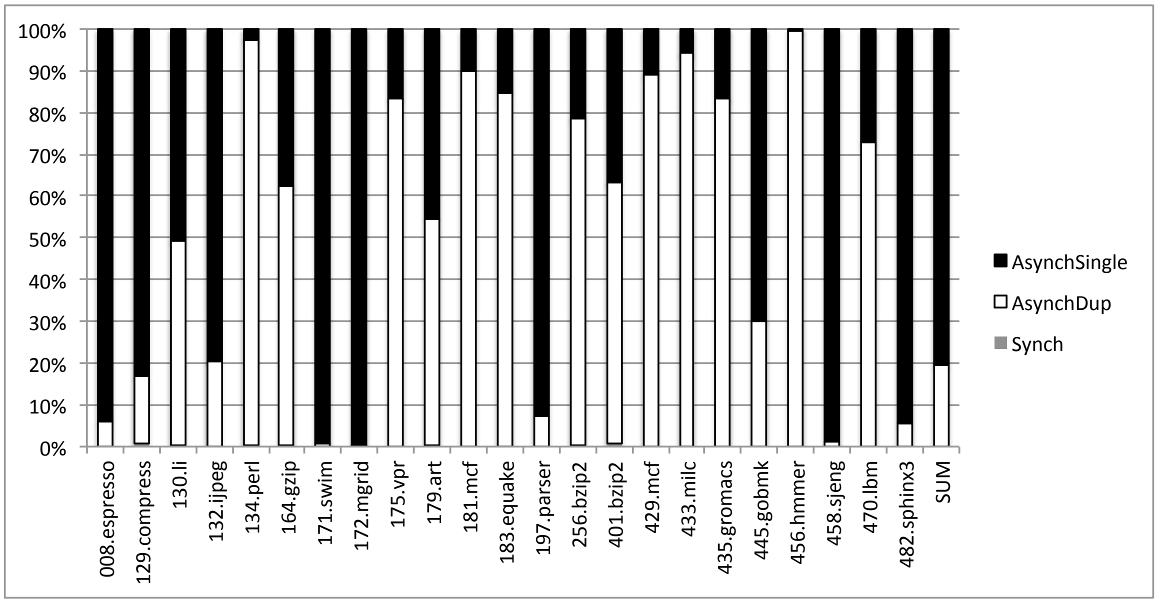


Figure 5.2: Normalized System Calls Categories in each benchmark

5.6 Performance Evaluation Methods

5.6.1 Runtime Overhead

The runtime overhead is one of the most important metrics used to evaluate RSFT. The runtime performance of RSFT is evaluated in comparison with the non-speculative execution. The runtime overhead is obtained from measuring the average execution time of RSFT from five runs on an empty unoccupied multi-core system, normalized to the original unprotected program execution time.

This evaluation was conducted using `time` on Linux. The execution time includes RSFT initialization, new process forking, checkpointing (if recovery is enabled), program computation, and the time spent in the operating system.

5.6.2 Memory consumption

Although duplicating a program execution doubles the amount of virtual memory used by the original program instance, it does not necessarily duplicate program's physical memory consumption. Compared with virtual memory size, extra physical memory usage is a more accurate metric in evaluating how much memory resources are consumed that can be used by other processes.

To compute how much physical memory is used at runtime by RSFT and the duplicated program instance, this dissertation instruments in program execution to evaluate real-time physical memory usage. This is achieved through stalling process execution periodically, then walking through the page table of stalled process. If a virtual memory page in page table has a corresponding physical page address, it is considered a page in memory. In addition, the tracing process itself is also periodically stalled and checked for physical memory consumption to get a total number for all processes.

5.6.3 Power consumption

Similar to memory consumption, duplicating program execution does not necessarily introduce more power consumption. The evaluation in this dissertation uses a power monitor attached to the evaluation platform. Its voltage is observed during program runtime to compute the power consumed by the whole evaluation platform. Since the evaluation is carried out on an unoccupied system, the power consumption is the power used by the hardware resources that execute the operating system, RSFT, and RSFT protected program.

The power consumption was measured in a separate process during the execution of RSFT protected benchmark programs. The measurement was done via sampling power voltage periodically, according to a pre-defined sampling rate. In this dissertation, the sampling rate is set to once per 10 second. The average voltage is computed as the average of all samples. The power consumption is proportional to the time spent in executing the program, multiplied by the average consumption per sample.

This dissertation report total power consumption of the entire machine, including CPU, memory, disks etc. Full system power was measured at the maximum sampling rate (13 samples per minute) supported by the power distribution unit (AP7892 [55]). Newer chips have power monitoring units with higher sampling rates and clock gating per core. They could be used to obtain more accurate power consumption in the future.

5.6.4 Transient Fault Recovery

The transient fault recovery scheme is evaluated using the following metrics : (1) Whether the faulty program execution can be recovered or not; (2) How much time is used to recover from a detected failure.

When a transient fault fail is detected, RSFT recovers the program execution using a previous verified checkpoint. Unlike previous solutions that must restart program from the very beginning, RSFT maintains the last verified checkpoint, and verifies again right before recovering from that checkpoint. However, due to the time spent on checkpointing and program state recovery, it is not free to recover from the fault. The mean time to recovery (MTTR) is also evaluated and reported in Chapter 6.

Chapter 6

Evaluation

6.1 Reliability

The reliability of fault tolerant techniques can be evaluated by their windows of vulnerability and mean time to fault detection. This section focuses on these two metric: window of vulnerability and mean time to detection. Both were measured and analyzed with register fault simulation and memory fault simulation.

6.1.1 Window of Vulnerability

To study the window of vulnerability of RSFT, a process tracing tool (`ptrace`) was used to simulate single-event-upset (SEU) [41, 45, 59] transient faults at runtime in both register files and the memory (in a way similar to [20, 50, 21]). Figure 6.1 shows the fault distribution of injected faults. The x-axis shows the faults injected into register files (R) or memory (M) for each benchmark program. The y-axis is the percentage of faults in each category.

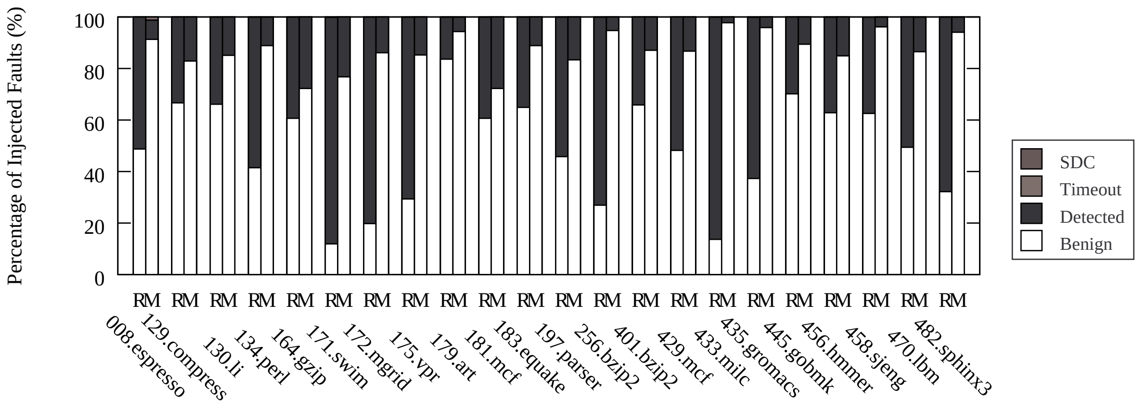


Figure 6.1: Register (R) and Memory (M) Transient Fault Distribution

Register Fault Tolerance

The R columns in Figure 6.1 shows the experimental results from injecting faults into the register file at program runtime. First, a profile run of the original program binary is timed to estimate how long it may take to execute the program. Before fault injection, our tool randomly selects one point in time, one random program instance, one random bit of a register, as well as one random register among general-purpose, floating point, XMM, and flag registers. During program runtime, this fault injection simulation issues an alarm after a random period of time. It then sends a signal to the randomly selected process, stops its execution, and flips the random bit of the selected register. The particular program instance then continues execution. In our evaluation, we only inject faults into the two redundant program instances, but not into the monitoring process. Since the monitoring process spends very little CPU time (CPU utilization is lower than 0.01%), it is extremely unlikely that a transient fault occurs while the monitoring process is executing. It must also be noted that this fault injection method does not inject faults into the operating system. Finally, the execution result of the fault-injected program is compared against the reference output to ensure that the RSFT-protected program's externally visible behavior is correct. Figure 6.1 shows the aggregated fault distribution over 3000 runs.

Benchmark	Occurrence for Register Injected Faults (%)				Occurrence for Memory Injected Faults (%)			
Name	Benign Faults	Detected Faults	Timeout	SDC	Benign Faults	Detected Faults	Timeout	SDC
008.espresso	48.74	51.23	0.03	0.0	91.32	7.47	1.21	0.0
129.compress	66.69	33.31	0.00	0.0	82.87	17.10	0.03	0.0
130.li	66.13	33.83	0.03	0.0	85.15	14.85	0.0	0.0
134.perl	41.50	58.46	0.03	0.0	88.90	11.06	0.04	0.0
164.gzip	60.70	39.23	0.07	0.0	72.27	27.73	0.0	0.0
171.swim	11.92	87.97	0.11	0.0	76.77	23.23	0.0	0.0
172.mgrid	19.83	80.10	0.07	0.0	86.13	13.87	0.0	0.0
175.vpr	29.36	70.60	0.04	0.0	85.27	14.70	0.03	0.0
179.art	83.66	16.25	0.09	0.0	94.35	5.61	0.04	0.0
181.mcf	60.70	39.23	0.07	0.0	72.27	27.73	0.0	0.0
183.equake	64.94	34.98	0.08	0.0	88.90	11.07	0.03	0.0
197.parser	45.76	54.17	0.07	0.0	83.37	16.60	0.03	0.0
256.bzip2	26.92	72.98	0.10	0.0	94.78	5.22	0.0	0.0
401.bzip2	65.89	34.11	0.0	0.0	87.10	12.90	0.0	0.0
429.mcf	53.23	46.74	0.03	0.0	55.33	44.67	0.0	0.0
433.milc	13.67	86.25	0.08	0.0	97.81	2.19	0.0	0.0
435.gromacs	37.30	62.57	0.13	0.0	95.87	4.10	0.03	0.0
445.gobmk	70.18	29.79	0.07	0.0	89.44	10.56	0.0	0.0
456.hmmer	62.80	37.13	0.10	0.0	84.90	15.10	0.0	0.0
458.sjeng	63.57	36.33	0.0	0.0	96.13	3.83	0.03	0.0
470.lbm	49.42	50.55	0.03	0.0	86.54	13.41	0.05	0.0
482.sphinx3	32.20	67.73	0.07	0.0	94.07	5.88	0.05	0.0

Table 6.1: Distribution of the outcomes of injected faults. The numbers correspond to 3000 runs of each benchmark.

The values for register fault injection in Figure 6.1 show that a large percentage of transient faults injected are benign faults, which do not result in either program crash or wrong output. Among the non-benign faults, all faults injected are detected by RSFT via either value comparison, customized signal handling, or timeout watchdog. Timeout occurs very rarely (0.1% on average). It is not visible in Figure 6.1 but is shown in Table 6.1.

Memory Fault Simulation

The instruction-level redundancy and redundant multi-threading approaches only maintain one memory state, and rely on ECC memory to protect programs against memory transient faults. Process-based redundancy techniques maintain multiple memory states and can provide transient fault protection for memory. However, prior work using process-based redundancy, including PLR, does not duplicate physical memory pages unless one or more values are written to that page. This reduces their fault coverage against transient faults in memory. This dissertation is simulates and evaluates the detection of transient faults in memory.

The M columns in Figure 6.1 and Table 6.1 demonstrate the experimental results from injecting faults into memory subsystems. Similar to injecting faults into register files, memory fault injection involves a profile run, and a random selection of a program point. Subsequently, memory fault injection randomly selects a virtual memory address owned by one random process, and randomly flips a bit of the value stored in that memory address, only if the page is physically in memory. This memory address may contain data or text of the program. Memory faults are injected into all running processes (the monitoring process, and two program instances), with the likelihood of the occurrence of the faults being proportional to the amount of memory used by each process. Each program was executed 3000 times with one memory fault injected each time.

Memory faults injected into RSFT itself may also be detected. For example, if a fault changes the value in memory that stores the system calls' arguments for later comparison, this fault will be detected when the values are compared against values from the other process later. Another example is that the memory fault injected may be transitively passed on to one of the processes and results in a value mismatch in later transient fault detection. However, if an injected memory fault changed a value in memory that is transitively passed on to both processes, and that value affects the final output, RSFT will not be able to detect it.

As shown in the Figure 6.1, memory faults usually do not result in erroneous program output. As observed in experiments, a large percentage of memory faults are benign faults. Only an average of 12.83% memory faults are non-benign faults and are detected by RSFT. For 179.art, the original program only consumes 10MB memory at runtime, therefore the probability of memory faults occurring in the monitoring process is as large as 50%. Experiment results shows that among 129 detected faults and 2 timeout out of 3000 runs, 16

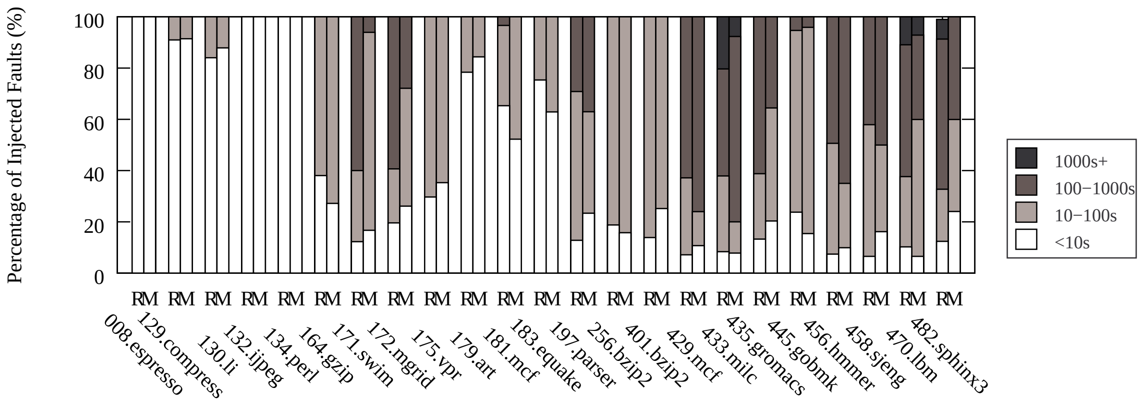


Figure 6.2: Mean time to detection for RSFT-NoSpec and RSFTwith and without fault recovery.

detected faults and 1 timeout are caused by memory faults in the monitoring process.

6.1.2 Mean Time to Detection

Because RSFT is a process-level redundant execution technique, low runtime overhead is achieved via infrequent value verification and system call interception. However, the low overhead also comes at a cost of delaying transient fault detection until it propagates to some value that escapes RSFT's SoR. To understand the delay of transient fault detection in RSFT, the mean time to detecting a fault is evaluated on all benchmark programs, as shown in Figure 6.2.

Figure 6.2 demonstrates that for most transient fault injected, RSFT is able to detect the faults within 10 seconds. 008.espresso, 134.perl and 164.gzip takes less than 10 seconds to complete, hence the mean time to detection is always less than 10 seconds. For programs that takes a long time to complete, such as SPEC2006 benchmark prorgrams, majority of transient faults, if not benign, are detected and reported within 300 seconds, or 5 minutes.

6.2 Performance

This section discusses the performance overhead of RSFT , with and without recovery. The overhead includes runtime overhead, resource consumption overhead, such as memory and power.

6.2.1 Runtime Overhead

Figure 6.3 shows the runtime overhead (vertical axis in the figure) of RSFT, normalized with respect to the original program execution without any transient fault detection. The result is compared with RSFT without speculation enabled. We also evaluated the performance overhead of dynamic instrumentation (PIN), which is the base of a previous related work PLR [49], on the same set of benchmarks. This is measured by executing the original program binary within the PIN framework on an unloaded machine, without any instrumentation. The average overhead of the framework alone is as high as 48.57%. Running redundant instances and runtime instrumentation on top of the framework, in addition to frequent synchronization, can only adds more runtime overhead. Compared with dynamic instrumentation approaches, such as PLR, RSFT provides the same applicability, but better coverage and much lower runtime overhead.

RSFT-NoSpec is implemented using barrier synchronizations at every system call, the same as what PLR does in Figure 2.4, but without the dynamic binary instrumentation overhead in PLR. This implementation does not speculate any program state after a system call is completed. Any program instance that issues a system call first stops program execution, waits until the other instance issues the same call, then starts executing the system call. Another synchronization is required upon returning from the system call execution in kernel. The return values of the system call is duplicated in both program instances at that point.

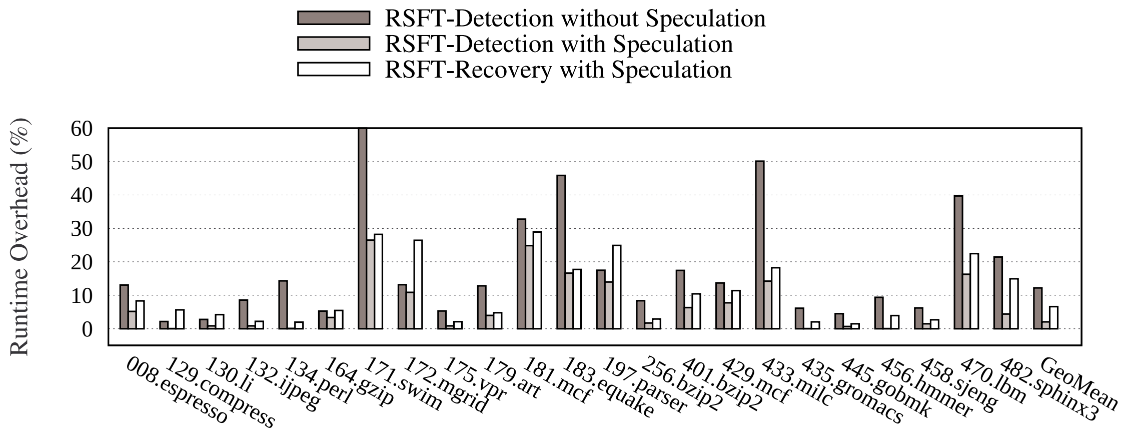


Figure 6.3: Performance overhead for RSFT-Detect with and without speculation, and RSFT-Recover

Some benchmarks, such as 183.equake, feature system calls in inner loops. This results in a large amount of inter-process barrier synchronization, if speculation is not available. Frequent synchronizations prevent overlapping of useful computation with fault checking, hence the synchronization overhead is placed on the critical path. As a result, running RSFT without speculation on 183.equake adds 45.86% overhead compared with the unprotected sequential program, while RSFT (with speculation) added only 16.60% performance overhead. Similarly, 171.swim and 433.milc share the same patterns, and gained huge performance improvement from speculation.

In this evaluation, RSFT never misspeculates because the system calls' return values are all correctly speculated. In cases where misspeculation occurs, it costs merely a couple of millisecond for the program to recover from a misspeculation.

6.2.2 Physical Memory Usage

As a technique that maintains multiple memory states, RSFT adds considerable memory consumption to the system. A program and its redundant copy together occupy twice as much physical memory as the original unprotected program. RSFT itself, however, only

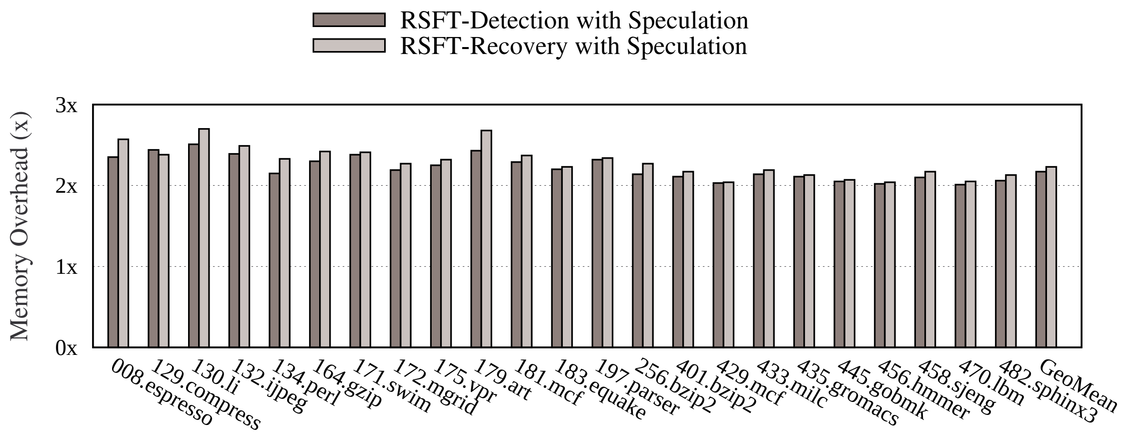


Figure 6.4: Physical memory overhead for RSFT with and without fault recovery.

maintains several static objects and dynamically allocates (and frees) some memory space to buffer data system call arguments for transient fault detection. It consumes around 9.5MB virtual memory at peak, since it is merely a thin layer between the applications and the underlying systems.

Figure 6.5 demonstrates the physical memory consumption measure at program runtime. The memory overhead of both RSFT-NoSpec and RSFT with and without fault recovery are measured and compared in this Figure as well.

Previous process-level redundant execution approaches, such as PLR, use triple-modular redundancy for fault recovery. As a result, their implementation must features three identical program instances in addition to the tracing process, adding even more overhead to memory consumption. Consequently, the implementation of RSFT introduces more memory overhead than the other implementations. Compared with fault recovery using triple redundancy and triple memory usage, RSFT provides a method to recover from transient faults fast, while only doubling the physical memory usage.

Because RSFT's random physical memory walking can be enabled and disabled at runtime, RSFT can use less physical memory at the cost of reduced fault coverage against memory transient faults.

Figure 6.5 indicates that the tracer program itself does introduce more physical memory usage to the system. Some benchmark programs have smaller memory footprint by itself. For example, 179.art only uses ttt physical memory pages at its peak. Because RSFT's tracing process itself uses mmm memory pages at runtime to keep its own data structure and stack, RSFT adds more memory pressure on programs like 179.art. Note that the software programs are becoming larger and more complex than ever with the evolve of modern architecture, the memory footprint of programs is also becoming bigger than ever. An example is that all SPEC 2006 benchmark programs consume much more memory than SPEC 2000 programs. Since the memory consumption of RSFT itself remains relatively constant, the memory pressure that RSFT adds to program execution will also become relatively smaller.

Although the memory consumption overhead seems to be high compared with the original memory usage, the real extra memory consumption, measured in meagbytes, is not significant. As the main memory is becoming an abundant resource in modern architecture, the extra memory used by RSFT is usually not a concern. Notice that using extra memory may introduce more swapping between main memory and the disk. This overhead is included in performance measured in Figure 6.3.

6.2.3 Power Consumption

The power consumption of RSFT with and without fault recovery is also evaluated in this dissertation for thorough understanding of the overhead of this technique. The computation in the redundant process and computation and the tracer process uses extra CPU cycles, hence increasing power usage as well.

However, the evaluation shows that the power consumption of RSFT protected pro-

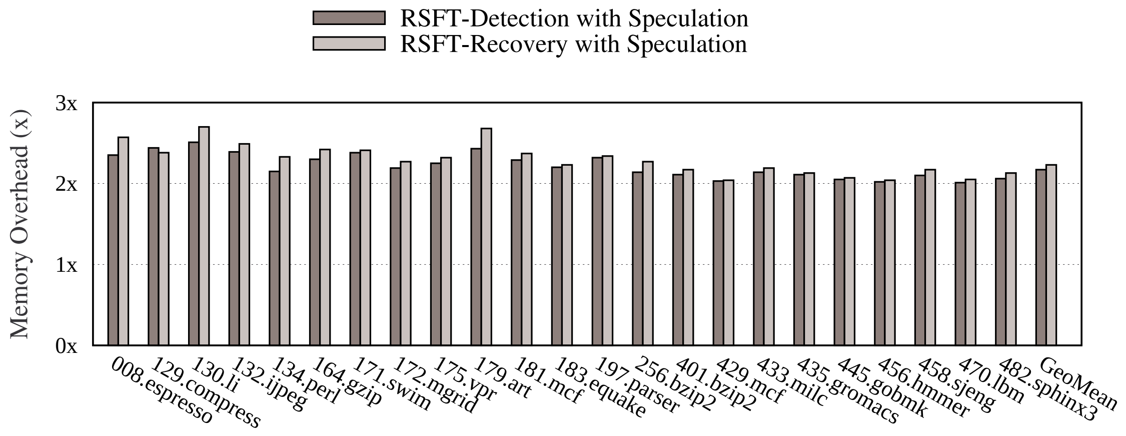


Figure 6.5: Physical memory overhead for RSFT with and without fault recovery.

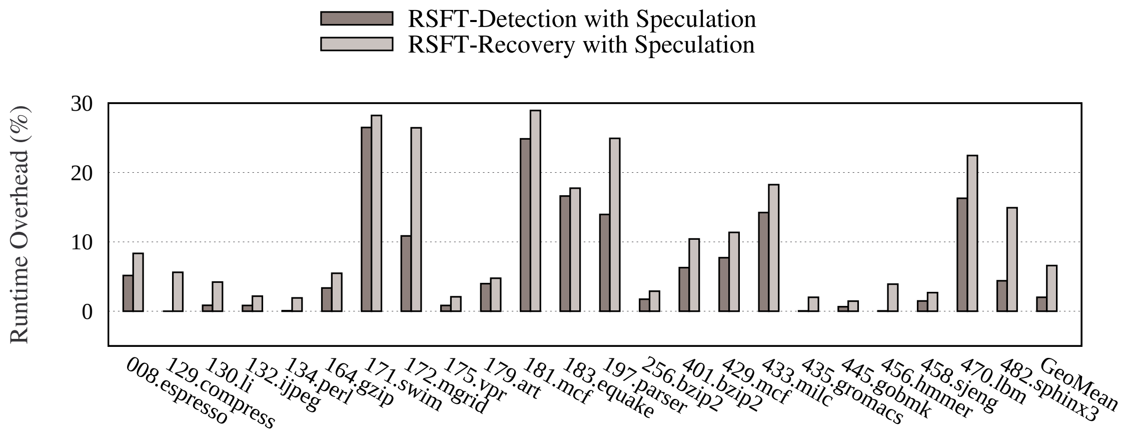


Figure 6.6: Power consumption overhead for RSFT with and without fault recovery.

grams is no more than the original program, using the evaluation method presented the Chapter 5. Figure 6.6 shows the power consumption of RSFT with and without fault recovery, normalized to the unprotected original program.

The experiments show that the power consumption at any given sampling moment remains the same for the original program, and the duplication execution protected by RSFT. As a result, the power consumption overhead is proportional to the runtime overhead of RSFT with and without recovery.

6.2.4 Checkpointing Overhead

The checkpointing method in RSFT involves very low runtime overhead. RSFT clones child processes from both worker processes and verifies if they are identical using a background checker. The original two worker processes also have to pay the cost of memory page copy-on-write. If any physical page in memory is written, the original worker processes will have to create another physical page in memory, potentially misses the cache.

The overhead of checkpointing is negligible if it is performed infrequently. However, in cases where frequent checkpointing is needed, the performance overhead can be higher than desired. Figure 6.7 shows how much performance overhead changes with the frequency of checkpointing.

In the performance evaluation above, RSFT performs checkpointing either every 5000 system calls, or once per run, whichever is greater.

6.2.5 Transient Fault Recovery Overhead

Another metric is how fast RSFT can recover a program from a transient fault caused failure. By using a process duplication and memory copy-on-write checkpointing method, transient fault recovery only involves waking up the checkpoint processes and continue execution from the point of last checkpoint. Hence recovery process is very fast in RSFT. Experiments show that recovering from a checkpoint process takes merely half a millisecond.

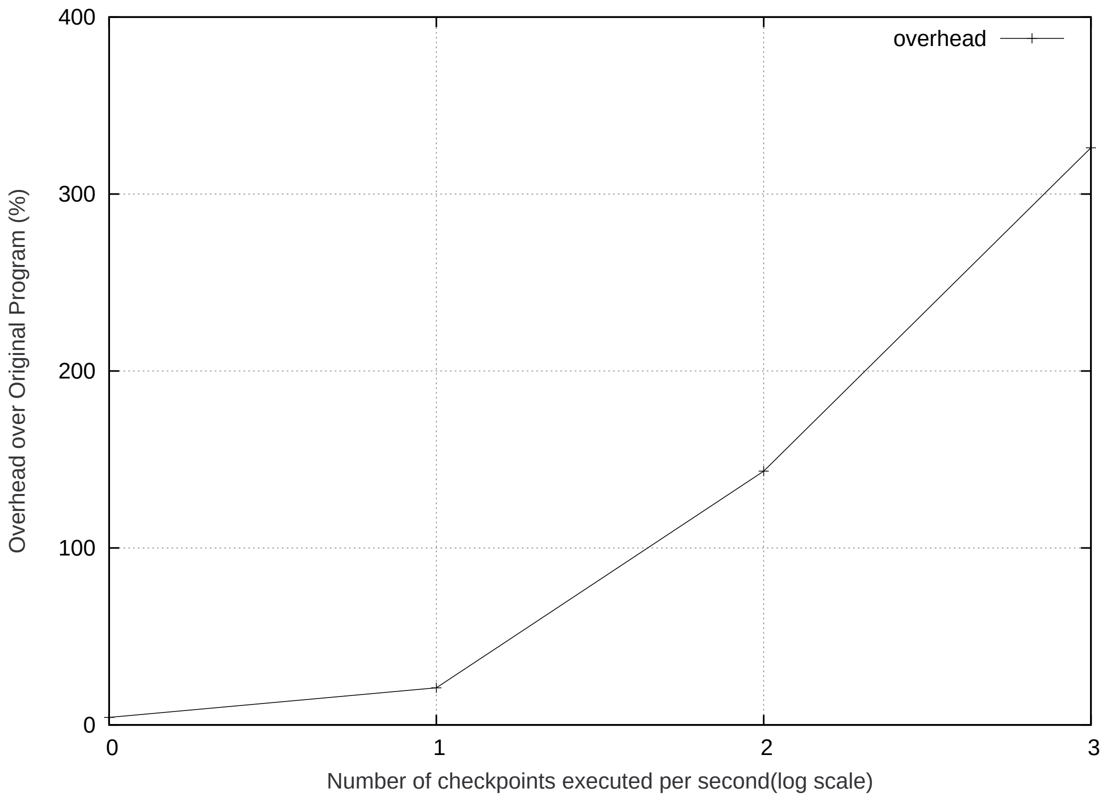


Figure 6.7: Checkpointing overhead for RSFT with and without fault recovery.

Chapter 7

Related Work

Chapter 2 introduced the most related transient fault tolerant techniques. This section discusses some additional related work.

DIVA [63] uses additional hardware checkers to provide fault protection. DieHard [6] proposed by Berger et al. uses redundancy on general-purpose machines for memory fault tolerance. Exterminator [33] uses process replicas to detect memory errors with high probability. RSFT also detects transient faults in memory, as well as register files.

Tapus et al. [53] introduce a syntax and an operational semantics for speculative execution for reliability and fault tolerance. This work proves that the speculative execution model is equivalent to the non-speculative model. Weaver et al. [64] and Vijaykumar et al. [58] use program behavior to direct transient fault detection. These techniques follow the propagation of faults through the program to reduce unnecessary replication. RSFT achieves the same goal by only detecting non-benign faults when they are about to affect program output. Gaiswinkler et al. [13] use the compiler to generate diverse binaries for a program, thus detecting a majority of faults that affect register values. However, this technique has a lower fault coverage than RSFT, and may produce false-positives. Aidemark et al. [1] propose methods to detect software errors by executing an application multiple times and majority voting.

Lee et al. propose Respec, an online multiprocessor replay technique using speculative logging for externally deterministic replay [24]. This technique optimistically logs less information about shared memory dependencies than needed for deterministic replay. If the replayed process diverges from recorded process, misspeculation recovery is performed. This paper and Respec follow the same idea that only externally visible behavior and final state of the program must be correct. Respec was introduced to replay externally visible behavior of shared memory multithreaded programs on commodity multiprocessor architecture. Similarly, RSFT can also be extended to provide transient fault detection for multithreaded programs with deterministic externally visible behavior.

Daniel et al. [11] explored the possibility of using process monitoring utilities of Unix systems for fault tolerance on cluster platforms with NFS. Process monitoring utilities are also exploited for related research topics, such as fault injection and program security checking [50, 21, 10, 20, 62]. Jarboui et al. present a software-implemented fault injection technique to identify different techniques for generating different software faults for operating systems [20]. In this work, they use the UNIX `ptrace` function to trap kernel calls issued by the process where the faults are being injected. Sieh implements a fault injector using the UNIX `ptrace` process monitoring interface that can inject transient faults into most of the CPU registers, FPU and FPA registers, and into the virtual address space of the running process [50]. FERRARI [21] uses the UNIX `ptrace` function to corrupt the memory image of a process at runtime. The `ptrace` function is used to insert software trap instructions at the specific instruction address where a fault is to be injected. FERRARI can inject faults into the data and code segments of a running process as well as the registers and part of the main memory used by that process. It can also intercept the system calls made by the running process, and change their return values. Buchacker et al. develop a framework for testing the fault-tolerance of systems, where they inject faults into a simulated system of Linux machines using the `ptrace` interface [10]. This paper uses similar methods to inject faults into register files and main memory to study the fault coverage of RSFT.

Chapter 8

Conclusion and Future Work

This dissertation presents a software-only speculative transient fault tolerance system called RSFT. This system advances the state-of-the-art transient fault tolerance techniques by achieving lowest runtime overhead and highest fault coverage. This chapter summarizes the work presented in previous chapters and concludes. Section 8.2 discusses potential future research on this topic.

8.1 Conclusions

Architectural trends toward smaller transistors, higher transistor counts, and lower core voltage make transient faults a more critical reliability concern than ever. Redundant hardware provides transient fault detection at the cost of additional chip area and design cost. Software redundancy is more appealing for its flexibility and low cost. However, even the best available software techniques for transient fault detection have large performance overhead.

This dissertation presents Runtime Speculative Software-only Fault Tolerance(RSFT), the fastest software fault detection technique to date. Combining OS-level process monitoring, value speculation, and efficient misspeculation recovery, RSFT provides transient fault tolerance with only 6.17% runtime overhead. With the flexibility of software and the low runtime overhead, RSFT provides a practical transient fault protection scheme for modern multicore processors.

This dissertation introduces a complete system for transient fault tolerance, including speculative fault detection, runtime lightweight checkpointing and background memory scrubbing for fault recovery. This system provides both the ease of use and high performance as well as full fault coverage. This dissertation also study and evaluate the effect of transient fault hitting a program's physical memory space.

8.2 Future Directions

Potential future research directions based on the contribution of this dissertation include configurable fault protection and runtime performance tuning. The software-only speculative fault tolerance techniques proposed in this dissertation provides full fault protection at the application level. For large software systems that do not require high reliability for all its components, it may be beneficial to protect only the critical code regions.

For example, the numerical computation of a bank's online banking system is very sensitive to transient faults and must be fault tolerant. However, the graphic interface itself of an online banking system is not. Missing one pixel or changing the color of one pixel on the screen is not critical and seldom noticed, as long as the fault does not crash the system. Different programs may need different level of protection, with different granularity as well. Hardware fault tolerant solutions will not be able to adjust its policy depending on which program is currently scheduled to run on a particular processor. Software solutions, especially runtime techniques like RSFT is able to adjust to different fault tolerance requirements after the system is deployed. This can be realized via either programmer specification, or runtime notification. The tracer process can only start fault tolerant execution once an environmental variable is set or a signal is received.

Another natural extension of this dissertation is applying the techniques described to distributed applications with non-deterministic outputs. This requires redefining faulty behavior of a program, as a program's redundant copy can behave differently from itself. One example is that a network application may send out heartbeat packets to the outside world. When a misspeculation occurs in one of the processes in RSFT, we have to know which system calls can be re-executed. This may require user level interference, such as pre-defining program behaviorin a way that RSFT can understand.

In the future, the methods use to evaluate and predict the reliability of a processor will be greatly appreciated. Recent research has attempted blah blah. These techniques can be used to dynamically determine the fault rate at runtime to adaptive configure checkpointing frequency as a response to the environmental changes.

Formal model checking methods have been proposed to verify transient fault tolerance techniques [37, 27, 23]. Proposals using symbolic execution or theorem proving have been applied to several existing fault tolerance techniques such as SWIFT. However, verification for binary level transient fault tolerance techniques still remains an open problem. Pattabiraman et al. [36] proposed SimPLIFIED, a symbolic fault injection framework for assembly level programs. However, this approach has high constraints. For example, SimPLIFIED only works for programs without floating point arithmetic, and is only applicable to programs with fault detection code embedded at assembly level. To verify and check for the correctness of RSFT, verification for multi-process programs must be provided.

8.3 Closing Remarks

Despite of all the efforts researchers made throughout the years, transient fault tolerance still remains one of the most difficult problems to solve. Among all the techniques proposed so far, hardware or software, most of them are far from being deployed in real commodity systems. The complexity and high cost of hardware approaches prevent their wide adoption. Existing software approaches, limited by either high performance penalty or limited fault coverage or limited applicability, still remain in research papers and are far from becoming practical.

This dissertation takes a step towards deploying a practical software method for transient fault tolerance, using the existing commodity hardware systems. Although RSFT has some window of vulnerability, but given its low overhead and high fault coverage, it has the potential of being applied to existing systems and provide transient fault protection at runtime.

Bibliography

[1] J. Aidemark, J. Vinter, P. Folkesson, and J. Karlsson. Experimental evaluation of time-redundant execution for a brake-by-wire application. In *DSN'02*, 2002.

[2] H. Ando, Y. Yoshida, A. Inoue, I. Sugiyama, T. Asakawa, K. Morita, T. Muta, T. Motokurumada, S. Okada, H. Yamashita, Y. Satsukawa, A. Konmoto, R. Yamashita, and H. Sugiyama. A 1.3GHz fifth generation SPARC64 Microprocessor. In *Digest of Technical Papers of the 2003 IEEE International Solid-State Circuits Conference*, 2003.

[3] H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, and C. Whelan. The sorcerer's apprentice guide to fault attacks. *Proceedings of the IEEE*, volume 94, pages 370–382, 2006.

[4] R. C. Baumann. Soft errors in advanced semiconductor devices: the three radiation sources. *IEEE Transactions on Device and Materials Reliability*, 2001.

[5] R. C. Baumann. Soft errors in commercial semiconductor technology: Overview and scaling trends. In *IEEE 2002 Reliability Physics Tutorial Notes, Reliability Fundamentals*, pages 121_01.1 – 121_01.14, April 2002.

[6] E. D. Berger and B. G. Zorn. Diehard: Probabilistic memory safety for unsafe languages. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2006.

[7] D. Boneh, R. A. DeMillo, and R. J. Lipton. On the importance of checking cryptographic protocols for faults. *Journal of Cryptology*, volume 14, pages 101–119, 2001.

[8] S. Borkar. Microarchitecture and design challenges for gigascale integration. In *MICRO'04*.

[9] G. Bronevetsky, B. R. de Supinski, and M. Schulz. A foundation for the accurate predication of the soft error vulnerability of scientific applications. In *Proceedings of the 5th Silicon Erros in Logic - System Effects*, 2009.

[10] K. Buchacker and V. Sieh. Framework for testing the fault-tolerance of systems including os and network aspects. In *Proceedings of the 6th IEEE International Symposium on High Assurance Systems Engineering*, 2001.

[11] E. Daniel and G. S. Choi. Tmr for off-the-shelf unix systems. In *Proceedings of the 29th International Symposium on Fault-Tolerant Computing*, 1999.

[12] S. Feng, S. Gupta, A. Ansari, and S. Mahlke. Shoestring: probabilistic soft error reliability on the cheap. In *ASPLOS'10*.

[13] G. Gaiswinkler and A. Gerstinger. Automated software diversity for hardware fault detection. In *ETFA '09: Proceedings of the 14th IEEE Conference on Emerging Technologies Factory Automation, 2009*, 2009.

[14] C. Gniady and B. Falsafi. Speculative sequential consistency with little custom storage. In *Proceedings of the 2002 International Conference on Parallel Architectures and Compilation Techniques*, pages 179–188, 2002.

[15] M. Gomaa, C. Scarbrough, T. N. Vijaykumar, and I. Pomeranz. Transient-fault recovery for chip multiprocessors. In *ISCA '03: Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 98–109, 2003.

[16] S. Govindavajhala and A. Appel. Using memory errors to attack a virtual machine. In *Proceedings of the 2003 Symposium on Security and Privacy*, pages 153–165, May 2003.

[17] S. Hareland, J. Maiz, M. Alavi, K. Mistry, S. Walsta, and C. Dai. Impact of cmos scaling and soi on software error rates of logic processes. *VLSI Technology Digest of Technical Papers*, 2001.

[18] R. W. Horst, R. L. Harris, and R. L. Jardine. Multiple instruction issue in the NonStop Cyclone processor. In *Proceedings of the 17th International Symposium on Computer Architecture*, 1990.

[19] T. B. Jablin, Y. Zhang, J. A. Jablin, J. Huang, H. Kim, and D. I. August. Liberty Queues for EPIC Architectures. In *Proceedings of the 8th Workshop on Explicitly Parallel Instruction Computing Techniques*, 2010.

[20] T. Jarboui, J. Arlat, Y. Crouzet, and K. Kanoun. Experimental analysis of the errors induced into linux by three fault injection techniques. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, 2002.

[21] G. A. Kanawati, N. A. Kanawati, and J. A. Abraham. Ferrari: A flexible software-based fault and error injection system. *IEEE Transactions on Computers*, 1995.

[22] T. Karnik, B. Bloechel, K. Soumyanath, V. De, and S. Borkar. Scaling trends of cosmic ray induced soft errors in static latches beyond 0.18. In *Proceedings of the Symposium on VLSI Technology*, pages 61–62, 2001.

[23] D. Larsson and R. Hhnle. Symbolic fault injection. In *International Verification Workshop*, 2007.

[24] D. Lee, B. Wester, K. Veeraraghavan, S. Narayanasamy, P. M. Chen, and J. Flinn. Respec: efficient online multiprocessor replayvia speculation and external determinism. In *ASPLOS'10*.

[25] M. Li, P. Ramach, S. K. Sahoo, S. V. Adve, V. S. Adve, and Y. Zhou. Swat: An error resilient system. In *Proceedings of the Fourth Workshop on Silicon Errors in Logic - System Effects*, 2008.

[26] A. Mahmood and E. J. McCluskey. Concurrent error detection using watchdog processors-a survey. *IEEE Transactions on Computers*, volume 37, pages 160–174, 1988.

[27] M. L. Meolad and D. W. Walker. Reasoning about fault tolerant programs. In *Proceedings of the European Symposium on Programming*, 2010.

[28] S. E. Michalak, K. W. Harris, N. W. Hengartner, B. E. Takala, and S. A. Wender. Predicting the number of fatal soft errors in los alamos national labratory's ASC Q computer. *IEEE Transactions on Device and Materials Reliability*, 2005.

[29] S. S. Mukherjee. Detailed design and evaluation of redundant multithreading alternatives. In *In Proceedings of the 29th Annual International Symposium on Computer Architecture*, 2002.

[30] S. S. Mukherjee, M. Kontz, and S. K. Reinhardt. Detailed design and evaluation of redundant multithreading alternatives. *SIGARCH Computer Architecture News*.

[31] S. S. Mukherjee, C. Weaver, J. Emer, S. K. Reinhardt, and T. Austin. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, 2003.

[32] E. Normand. Single event upset at ground level. *IEEE Transactions on Nuclear Science*, volume 43, pages 2742–2750, Dec 1996.

[33] G. Novark, E. D. Berger, and B. G. Zorn. Exterminator: automatically correcting memory errors with high probability. In *PLDI'07*.

[34] T. J. O'Gorman, J. M. Ross, A. H. Taber, J. F. Ziegler, H. P. Muhlfeld, I. C. J. Montrose, H. W. Curtis, and J. L. Walsh. Field testing for cosmic ray soft errors in semiconductor memories. In *IBM Journal of Research and Development*, 1996.

[35] N. Oh, P. P. Shirvani, and E. J. McCluskey. Error detection by duplicated instructions in super-scalar processors. In *IEEE Transactions on Reliability*, volume 51, pages 63–75, March 2002.

[36] K. Pattabiraman, N. Nakka, Z. Kalbarczyk, , and R. Iyer. Symplfied: Symbolic program-level fault injection and error detection framework. In *Proceedings of International Conference on Dependable Systems and Networks*, 2010.

[37] F. Perry, L. Mackey, G. A. Reis, J. Ligatti, D. I. August, and D. Walker. Fault-tolerant typed assembly language. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2007.

[38] R. Phelan. Addressing soft errors in ARM core-based SoC. ARM White Paper, December 2003.

[39] M. Prvulovic, Z. Zhang, and J. Torrellas. Revive: Cost-effective architectural support for rollback recovery in shared-memory multiprocessors. In *Proceedings of the 29th annual international symposium on Computer architecture*, May 2002.

[40] A. Raman, H. Kim, T. R. Mason, T. B. Jablin, and D. I. August. Speculative Parallelization Using Software Multi-threaded Transactions. In *ASPLOS'10*.

[41] S. K. Reinhardt and S. S. Mukherjee. Transient fault detection via simultaneous multithreading. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, 2000.

[42] G. A. Reis. *Software Modulated Fault Tolerance*. PhD thesis, Department of Electrical Engineering, Princeton University, Princeton, New Jersey, 2008.

[43] G. A. Reis, J. Chang, D. I. August, R. Cohn, and S. S. Mukherjee. Configurable transient fault detection via dynamic binary translation. In *Proceedings of the 2nd Workshop on Architectural Reliability*, 2006.

[44] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August. SWIFT: Software implemented fault tolerance. In *Proceedings of the 3rd International Symposium on Code Generation and Optimization*, 2005.

[45] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, D. I. August, and S. S. Mukherjee. Design and evaluation of hybrid fault-detection systems. In *ISCA'05*.

[46] E. Rotenberg. AR-SMT: A microarchitectural approach to fault tolerance in microprocessors. In *Proceedings of the 29th Annual International Symposium on Fault-Tolerant Computing*, 1999.

[47] N. Saxena and E. J. McCluskey. Dependable adaptive computing systems – the ROAR project. In *International Conference on Systems, Man, and Cybernetics*, pages 2172–2177, October 1998.

[48] J. Segura and C. F. Hawkins. *CMOS Electronics: How It Works, How It Fails*. Wiley-IEEE Press, April 2004.

[49] A. Shye, T. Moseley, V. J. Reddi, J. B. t, and D. A. Connors. Using process-level redundancy to exploit multiple cores for transient fault tolerance. In *DSN'07*.

[50] V. Sieh. Fault-injector using unix ptrace interface. In *Internal Report 11/93, IMMD3, Universitt ErlangenNrnberg*, 1993.

[51] T. J. Slegel, R. M. Averill III, M. A. Check, B. C. Giamei, B. W. Krumm, C. A. Krygowski, W. H. Li, J. S. Liptay, J. D. MacDougall, T. J. McPherson, J. A. Navarro, E. M. Schwarz, K. Shum, and C. F. Webb. IBM's S/390 G5 Microprocessor design. In *IEEE Micro*, volume 19, pages 12–23, March 1999.

[52] J. C. Smolens, B. T. Gold, J. Kim, B. Falsafi, J. C. Hoe, and A. G. Nowatzyk. Fingerprinting: Bounding soft error detection latency and bandwidth. In *Architectural Support for Programming Languages and Operating Systems*, October 2004.

[53] C. Tapus and J. Hickey. Distributed speculative execution for reliability and fault tolerance: an operational semantics. 2009.

[54] T.C. and M. H. Woods. Alpha-particle-induced soft errors in dynamic memories. *IEEE Transactions on Electron Devices*, volume ED-26, page 2, 1979.

[55] The Metered rack PDU AP7892. Web site: http://www.apc.com/resource.

[56] Y. Tosaka, S. Satoh, K. Suzuki, T. Sugii, H. Ehara, G. Woffinden, and S. Wender. Impact of cosmic ray neutron induced soft errors on advanced submicron CMOS

circuits. In *Proceedings of the Symposium on VLSI Technology*, pages 148–149, June 1996.

[57] M. Tremblay and Y. Tamir. Support for fault tolerance in VLSI processors. volume 1, pages 388–392, May 1989.

[58] T. N. Vijaykumar, I. Pomeranz, and K. Cheng. Transient-fault recovery using simultaneous multithreading. In *Proceedings of the 29th ISCA*, 2002.

[59] D. Walker, L. Mackey, J. Ligatti, G. A. Reis, and D. I. August. Static typing for a faulty lambda calculus. *SIGPLAN Notices*, 2006.

[60] C. Wang, H.-S. Kim, Y. Wu, and V. Ying. Compiler-managed software-based redundant multi-threading for transient fault detection. In *CGO '07: Proceedings of the International Symposium on Code Generation and Optimization*, pages 244–258, Washington, DC, USA, 2007.

[61] N. J. Wang, J. Quek, T. M. Rafacz, and S. J. patel. Characterizing the effects of transient faults on a high-performance processor pipeline. *Dependable Systems and Networks, International Conference on*, volume 0, page 61, 2004.

[62] R. N. M. Watson. Exploiting concurrency vulnerabilities in system call wrappers. In *Proceedings of the first USENIX workshop on Offensive Technologies*, 2007.

[63] C. Weaver and T. M. Austin. A fault tolerant approach to microprocessor design. In *DSN'01*.

[64] C. Weaver, J. Emer, S. S. Mukherjee, and S. K. Reinhardt. Techniques to reduce the soft error rate of a high-performance microprocessor. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, 2004.

[65] Y. Yeh. Triple-triple redundant 777 primary flight computer. In *Proceedings of the 1996 IEEE Aerospace Applications Conference*, pages 293–307, February 1996.

[66] Y. Zhang, S. Ghosh, J. W. Lee, J. Huang, S. Mahlke, and D. August. Runtime asynchronouse fault tolerance via speculation. In *CGO'12: Proceedings of the International Symposium on Code Generation and Optimization*.

[67] Y. Zhang, J. W. Lee, N. P. Johnson, and D. I. August. Daft: Decoupled Acyclic Fault Tolerance. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, 2010.

[68] J. F. Ziegler, H. W. Curtis, H. P. Muhlfeld, C. J. Montrose, B. Chin, M. Nicewicz, C. A. Russell, W. Y. Wang, L. B. Freeman, P. Hosier, L. E. LaFave, J. L. Walsh, J. M. Orro, G. J. Unger, J. M. Ross, T. J. O'Gorman, B. Messina, T. D. Sullivan, A. J. Sykes, H. Yourke, T. A. Enger, V. Tolat, T. S. Scott, A. H. Taber, R. J. Sussman, W. A. Klein, and C. W. Wahaus. IBM experiments in soft fails in computer electronics (1978 - 1994). *IBM Journal of Research and Development*, volume 40, pages 3–18, January 1996.