FROM EXPERIMENT TO DESIGN – FAULT CHARACTERIZATION AND DETECTION IN PARALLEL COMPUTER SYSTEMS USING COMPUTATIONAL ACCELERATORS

BY

KEUN SOO YIM

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2013

Urbana, Illinois

Doctoral Committee:

Professor Ravishankar K. Iyer, Chair
Professor Lui R. Sha
Professor Roy H. Campbell
Professor Tarek Abdelzaher
Dr. Shuo Chen, Microsoft Research

Abstract

This dissertation summarizes experimental validation and co-design studies conducted to optimize the fault detection capabilities and overheads in hybrid computer systems (e.g., using CPUs and Graphics Processing Units, or GPUs), and consequently to improve the scalability of parallel computer systems using computational accelerators. The *experimental validation* studies were conducted to help us understand the failure characteristics of CPU-GPU hybrid computer systems under various types of hardware faults. The main characterization targets were faults that are difficult to detect and/or recover from, e.g., faults that cause long latency failures (Ch. 3), faults in dynamically allocated resources (Ch. 4), faults in GPUs (Ch. 5), faults in MPI programs (Ch. 6), and microarchitecture-level faults with specific timing features (Ch. 7). The *co-design* studies were based on the characterization results. One of the co-designed systems has a set of source-to-source translators that customize and strategically place error detectors in the source code of target GPU programs (Ch. 5). Another co-designed system uses an extension card to learn the normal behavioral and semantic execution patterns of message-passing processes executing on CPUs, and to detect abnormal behaviors of those parallel processes (Ch. 6). The third co-designed system is a co-processor that has a set of new instructions in order to support software-implemented fault detection techniques (Ch. 7).

The work described in this dissertation gains more importance because heterogeneous processors have become an essential component of state-of-the-art supercomputers. GPUs were used in three of the five fastest supercomputers that were operating in 2011. Our work included *comprehensive fault characterization* studies in CPU-GPU hybrid computers. In CPUs, we monitored the target systems for a long period of time after injecting faults (a temporally comprehensive experiment), and injected faults into various types of program states that included dynamically allocated memory (to be spatially comprehensive). In GPUs, we used fault injection studies to demonstrate the importance of detecting silent data corruption (SDC) errors that are mainly due to the lack of fine-grained protections and the massive use of fault-insensitive data. This dissertation also presents *transparent fault tolerance* frameworks and techniques that are directly applicable to hybrid computers built using only commercial off-the-shelf hardware components.

This dissertation shows that by developing understanding of the failure characteristics and error propagation paths of target programs, we were able to create fault tolerance frameworks and techniques that can quickly detect and recover from hardware faults with low performance and hardware overheads.

Acknowledgments

It has been about twenty years since I first started to study computer programming, science, and engineering. I would like to express my deepest gratitude to everyone who has inspired and helped me to learn and love computing.

First and foremost, I would like to deeply thank my thesis advisor, Professor Ravishankar K. Iyer, for his insightful research direction and generous support during my four and a half years of Ph.D. studies. The research methodology established over several decades by Ravi and his group forms the basis of my dissertation. I would also like to thank Dr. Zbigniew Kalbarczyk for teaching me about fault tolerance computing, and for many in-depth discussions.

I would like to thank all the members of my doctoral committee: Professor Lui R. Sha for teaching me about safety-critical system design; Professor Roy H. Campbell for teaching me about computer security; Professor Tarek Abdelzaher for teaching me about exergy-efficient design; and Dr. Shuo Chen for valuable discussions and feedback.

My doctoral research has taken advantage of collaborations with many outstanding researchers and engineers. I would like to thank Daniel Chen, Cuong Pham, Valentin Sidea, and all other members of Prof. Iyer's group for collaborations; and many CS and ECE graduate students for collaborations in classes at UIUC. I would also like to thank all the professors who have taught me, advised me, or provided feedback on my Ph.D. research. Underlying aspects of my research have been adjusted and/or stimulated through many meetings and discussions I had with people I met at schools, conferences, and workplaces. I would like to express my gratitude to all of them as well as all members of Escape; members of UIUC KCS; administrative staffs of CSL, ITI, and CSD; and the undergraduate students who paid attention to my lectures in the operating system class and my undergraduate mentees who have conducted research projects as part of the PURE program at UIUC.

Finally, I would like to share the delight of completing this dissertation as a milestone of my research journey with my respected parents, lovely wife, and proud children. I thank my respectful parents and parents-in-law for their love and support. I thank my lovely wife for her consistency and for always being with me. Without the devotion and patience of Jung Hyun, this dissertation would have taken much longer. I am proud of my faithful and honest daughters, and thank them for their existence and the happiness they create. The smiles of Sahngwie and Sahngyu were my fuel during my Ph.D. study.

To my parents, wife and children

To my teachers

Disclaimer

This material is based upon work supported by the National Science Foundation under Grant No. CNS-0524695. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

This material is based upon work supported by the Department of Energy under Award Number DE-OE0000097. This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

Table of Content

List of Figures

List of Tables

Nomenclature

ALU	Arithmetic Logic Unit
API	Application Programming Interface
APIC	Advanced Programmable Interrupt Controller
ASIP	Application Specific Instruction set Processor
ASIC	Application-Specific Integrated Circuit
CISC	Complex Instruction Set Computer
COTS	Commercial Off-The-Shelf
CUDA	Compute Unified Device Architecture ™
DEC-TED	Double Error Correction and Triple Error Detection
DFA	Deterministic Finite Automaton
DMA	Direct Memory Access
DRAM	Dynamic Random Access Memory
ECC	Error Correction Code
FDX	Fault Detection eXtension (Proper Noun)
EFI	Extensible Fault Injection (Proper Noun)
ELF	Executable and Linkable Format
ES	Error Sensitivity
FI	Fault Injection
FIT	Failures In Time per billion operation hours
FLOPS	Floating-Point Operations Per Second
FP	Floating Point
FPGA	Field Programmable Gate Array
FPU	Floating-Point Unit
FS	Fault Sensitivity
GPGPU	General-Purpose computing on Graphics Processing Unit
GPU	Graphics Processing Unit
GUI	Graphical User Interface
HDL	Hardware Description Language
HPC	High Performance Computing
IPMI	Intelligent Platform Management Interface
IR	Intermediate Representation
ISA	Instruction Set Architecture
JVM	Java Virtual Machine

KIPS	Kilo Instructions Per Second
LRU	Least Recently Used
NFA	Nondeterministic Finite Automaton
NIC	Network Interface Card
NMI	Non-Maskable Interrupt
NMR	N Modular Redundancy
MMU	Memory Management Unit
MMX	MultiMedia, Multiple Math, or Matrix Math eXtension ™
MPI	Message Passing Interface
MTTF	Mean Time To Failure
MUX	Multiplexer
OpenCL	Open Computing Language
OS	Operating System
PC	Program Counter
PCB	Process Control Block
PCI	Peripheral Component Interconnect
QoS	Quality-of-Service
SDC	Silent Data Corruption
SDR	Single Data Rate
SEC-DED	Single Error Correction and Double Error Detection
SEL	Single Event Latch-up
SEU	Single Event Upset
SPMD	Single Program Multiple Data
SMP	Symmetric Multi-Processing
SMT	Simultaneous Multi-Threading
SWIFI	Software-Implemented Fault Injection
TLB	Translation Lookaside Buffer
TMR	Triple Modular Redundancy
TOCTTOU	Time-of-Check-to-Time-of-Use
VLIW	Very Long Instruction Word
VM	Virtual Machine
VMM	Virtual Machine Monitor
XOR	Exclusive OR
3D	Three Dimensional Space

Chapter 1. Introduction

This chapter describes the reliability problem of parallel computer systems, and gives an overview of this dissertation's work on that problem.

1.1. Problem

Use of heterogeneous processing engines is a trend that attempts to optimize the computational throughput vs. power consumption and the computational throughput vs. hardware cost. Designers can use special-purpose processing units that are heavily customized for the characteristics of target application program workloads. For example, three out of the five fastest supercomputers in operation in 2011 (i.e., Tianhe-1A, Nebulae, and TSUBAME 2.0) use both CPU and GPU devices. Modern GPUs are a compelling platform for executing HPC programs (e.g., science simulation and medical data processing) because GPUs can process large volumes of data by using many collaborating threads. They can do so through use of multiple cores, wide memory bandwidths, large register files, and many arithmetic units. Those rich hardware resources lessen structural hazards. The throughput-driven design of the GPU core architecture addresses data and control hazards (i.e., the main barriers to exploitation of the instruction-level parallelisms in CPU designs). The direct exposure of GPU hardware resources to programmers via the programming model allows programmers to fully optimize their programs.

Early adapters of such special-purpose hardware take the risk of using newly developed hardware components. The development history of large-scale computers reveals such risks. The ASC Q supercomputer at Los Alamos National Laboratory (LANL) experienced about 27.7 failures per week because of faults in its processors [MHH+05]. Most of these failures were due to faults in the processor on-chip caches. The on-chip caches were protected by a parity-based error detection technique without error correction capabilities. The problem was addressed in later generations of machines through introduction of an SEC-DED ECC technique in processor on-chip caches except for the first-level caches. Similarly, newly designed special-purpose processors can have unknown and unprotected reliability problems. That shows the need to perform experimental validation of such new hardware components and to use the experimental validation data to redesign fault tolerance techniques for such new components.

1.2. Objective

The main objective of this dissertation is to develop fault tolerant systems capable of handling computational intensive parallel applications, e.g., large-scale simulations.

(i) Efficiency. The first goal is to shift the scaling ceiling of parallel computer systems as much as possible by managing hardware faults in an efficient and organized manner. An efficient fault tolerance technique would have high fault detection and recovery coverage, and low runtime performance, memory space, and hardware area overheads.

(a) High coverage. Crash failure detection of a user process or OS in a single-node system is free and done by default, because such failures imply faults that are detectable by the baseline error detection techniques in the hardware, OS, middleware, or application. Our research thus focuses on detection of hang, data omission, timing, and data corruption failures of system and application software. Although those four types of failures are relatively rare, missing them has large penalties in large-scale computer systems in terms of performance and energy efficiencies. We use the following three metrics to evaluate fault detection and recovery coverage.

- *Detection coverage*. Detection coverage is defined as one minus the probability that a random hardware fault will go undetected by the used error detection techniques and lead to user-visible failures. Our target detection coverage is >99% for permanent faults, and >95% for transient faults in any software.
- *Recovery coverage*. Recovery coverage is defined as an average probability of tolerating detected errors with the used error recovery techniques. Our target error recovery coverage is >99% for errors and failures in CPUs and GPUs.
- *Detection latency*. Detection latency is the time from the activation of a hardware fault to the detection of the activated fault by the header node of a parallel or distributed system. Our target average detection latency is 1 second for crash failures, 10 seconds for hang failures, and 1 minute for silent data corruption (SDC)[1] failures. Note that this detection latency definition includes the error latency (the time from the activation of a fault to the detection of an induced failure by software on the same computer node), and its average value for crash failures is small in modern computer systems, according to previous experimental studies.

[1] An *SDC error* is defined as an undetected data error that violates the output data correctness of a target program. Its exact definition depends on the quality-of-service (QoS) requirements of an individual program and/or user.

(b) Low overhead. Computation, hardware cost, and energy efficiencies are key design requirements in any computer system. Error detection and recovery shall not harm any of those efficiencies in a noticeable way. We use the following three overhead metrics:

- *Performance overhead*. Performance overhead is a ratio of the execution time of a program with the presented fault tolerance techniques, and the execution time of the same program without the presented techniques. The two execution times are measured in fault-free conditions (e.g., in a small-scale system). Our target performance overhead is <1% for fault detection and <10% for fault recovery.
- *Memory space overhead*. Memory space overhead is a ratio of the size of extra memory space needed to run the presented fault tolerance techniques and the total size of used memory space of a target system without the presented techniques. The target memory overhead is <3% and is <50MB per user thread.
- *Hardware area overhead*. Hardware area overhead is the ratio of extra hardware area size needed to run the presented fault tolerance techniques and the hardware area size of a target system (i.e., excluding main memory size) without the presented techniques. Our target hardware overhead is <3%.

Moreover, the performance of tightly coupled parallel programs is highly sensitive to small performance jitters (i.e., delay made by underlying software and hardware), especially when these programs run at a massively large scale. Because hardware faults are common in large-scale systems, performance jitters due to detected and corrected faults have a non-negligible impact. That implies that fault detection and recovery techniques must not become another source of noticeable performance jitter, regardless of the presence and frequency of hardware faults in the system.

(ii) Transparency. The second goal consists of realizing the first goal in computer systems that were built from COTS hardware and software components. That means that the designed fault tolerance techniques must not require any intrusive modification to the target system and must be easy to deploy (e.g., pluggable architecture), regardless of whether the techniques are implemented in hardware or software.

Our research focuses on minimizing changes in the hardware and software of the target system (i.e., a *non-intrusive* design). For hardware techniques, a chip-level modification is less desired than a board- or system-level change. For software techniques, while software changes are preferred to hardware changes, it would be preferable to use an automatic binary translation rather than rely on source code annotation.

All of our presented techniques were designed to be easily installable by administrators or users in COTS-based target systems, without requiring intrusive modification of

the target systems. The components that form the presented techniques include an extension card, an OS kernel module, a user-level library, and a source-to-source translator. All of them can be plugged into the target system software or hardware. A normal user can compile and run his or her program by using the compilers, libraries, middleware, and OS installed in a target computer by administrators. The build and launch processes of the user will be no different from those used for other supercomputers or cloud servers. That shows the transparency of the presented techniques to normal users.

1.2.1. Assumption

We target parallel computer systems that use computational accelerators. We specifically target a hybrid computer system that uses CPUs (as main processors) and GPUs (as computational accelerators). Although we assume this specific combination of heterogeneous processor types, the presented techniques are generic, that is, are applicable to other hybrid computers to the extent that their computational acceleration processors are managed the way GPUs are managed in our target systems.

In our target system, a GPU program consists of CPU- and GPU-side codes. A GPU kernel is a part of the GPU-side code with an entry function callable from the CPU-side code. GPU kernels form a majority of total computation time in GPU-based parallel programs. The loop portions of GPU kernels typically dominate the GPU kernel execution time, because the degrees of parallelism of the software are higher than those available in the GPU hardware. Each GPU kernel uses the SPMD (Single Program Multiple Data) programming model.

GPUs are used as computational accelerators. Thus, the microarchitectures of GPUs are heavily optimized to provide high computation and energy efficiencies during processing of computation- and/or data-intensive programs. GPUs have many arithmetic units, large-size registers, and both hardware and software-managed on-chip caches. We assume that in the accelerators, computation operations are faster and more energy-efficient than memory operations are (i.e., there is a high processor-memory performance gap). On the other hand, the accelerators do not support fine-grained address translation (e.g., a translation lookaside buffer with page-granularity translation) or strong fault isolation between parallel threads (e.g., due to the shared memory model between parallel threads).

We assume that a parallel programming model is used to coordinate multiple GPU programs. The assumed parallel programming model is MPI (Message Passing Interface). A user located in the header node launches multiple MPI processes on top of multiple compute nodes, where each compute node has a set of main processors and accelerators.

Each MPI process repeatedly executes similar computation operations during the majority of its execution time. In each compute node, only one MPI program runs at a time (except for system daemons).

1.2.2. Definition

For the purposes of this dissertation, the definitions of *fault*, *error*, and *failure* are as follows [Lap95]. A *fault* is the root cause of a problem. An *error* is a state of a system changed or corrupted by a fault. A *failure* is a symptom or behavior of a system that has been exposed or revealed to a user. Depending on the perspective of that person, the actual phenomena that represent faults, errors, or failures vary. Each chain of a fault, error, and failure is connected to other chains in the system, where a failure in a chain (or an abstraction) can be a fault in another chain (i.e., in a higher layer of system abstraction or another system component). In this dissertation, we mainly use transient hardware fault models. The exact fault models used are discussed in Section 2.1.

Failures are classified into the following types: (a) *not activated*, meaning that a faulty state is not accessed during the computation; (b) *overwritten*, meaning that a faulty state is accessed, but the access operation is a write that overwrites a new value to the faulty state; (c) *system crash*, meaning that an activated fault is detected by the baseline error detector and stops the system software execution; (d) *system hang*, meaning that an activated fault is not detected but makes the system software malfunction; (e) *user crash*, meaning that an activated fault is detected and causes an unexpected termination of a user process; (f) *user hang*, meaning that an activated fault makes a user process run more than *n* times the expected execution time; (g) *user silent data corruption (SDC)*, meaning that an activated fault causes a corruption of application output data that violates the correctness requirement of the application; and (h) *non-manifested*, meaning that an activated fault causes none of the above-described failures. In this dissertation, a *benign fault* means an un-activated, overwritten, or non-manifested fault.

1.3. Contribution

This dissertation shows that *by developing understanding of the failure characteristics and error propagation paths of target programs, we were able to create fault tolerance frameworks and techniques that can quickly detect and recover from hardware faults with low performance and hardware overheads*. The experimental aspect of this dissertation shows that by emulating errors in high layers of system abstractions, one can conduct

reliability validation experiments on top of commodity computer systems in a nonintrusive way, and efficiently quantify system characteristics under hardware faults. The experimental part also shows that the selection of targets for protection should be based on knowledge of recoverability rather than on error sensitivity alone. The part focusing on design demonstrates that by developing intelligence on the normal behavioral and semantic execution patterns of application programs, external checkers can accurately and quickly detect non-benign hardware faults with low performance and hardware overheads.

1.3.1. Experiment

We developed a set of validation frameworks and measurement techniques, and performed dependability benchmarking experiments.

(i) Extensible fault injection framework. If the characteristics of computer systems can be accurately measured under various types of faults, this can give valuable insights into the design and operation of dependable computer systems. However, observations of a transient fault and resulting errors are difficult in real systems and often incur large overheads. Detecting a transient fault or an error event itself either requires a hardware modification (e.g., duplication) or causes a large performance overhead.[2] The reason is that it is necessary to monitor the system state continuously in order to detect a fault or an error. Here, the measurement precision depends on the monitoring frequency of a target system, while the monitoring overhead is directly proportional to the monitoring frequency and the size of monitored state. This trade-off makes an accurate detection of a transient fault or an error impractical in real systems.

Fault injection experiments enable accurate observations of fault, error, and failure events. Because a synthetic fault is injected into a target system, the experimenter knows when and where a fault is injected (i.e., the same as detecting the fault occurrence event). Using a fault injection tool, an experimenter can monitor any access to a specific system state that was corrupted by the injected fault (i.e., detection of an error event). That access monitoring does not introduce a large runtime overhead, because modern processors support a hardware breakpoint mechanism that raises an interrupt when a predetermined target memory state is being accessed (i.e., executed, read, or written). The performance

[2] Note that it is true that faults that have the same characteristics as transient faults are still diagnosable by off-line testing (e.g., BIST) in practice. For example, some memory or disk transient faults are related to the wear-outs or other permanent defects of the devices (e.g., a memory cell with a stuck-at-‘0’ fault causes an error at runtime only if the value of ‘1’ is written in it). Such off-line testing does not cause any runtime overhead, although this off-line testing itself can still takes a long time.

overhead directly caused by this hardware breakpoint is just a time to set a breakpoint and a time to process this interrupt request.

Several fault injection frameworks have been developed for quantitative evaluation of the dependability of mission-critical systems and high availability computer systems. However, it is still an open problem that is making an agreement on specific fault injection experimental methodology. Such an agreement would make it easy for other experimenters to repeat fault injection experiments. The problem is mainly due to the use of different fault injection tool, which is customized for specific experimental objectives of an individual or a group. Some researchers have applied a framework for various fault injection experiments (e.g., using different target systems or fault models). Such a framework can be extended efficiently to the extent that the original developers are involved in the extension. However, another experimenter who does not have sufficient information about the internal architecture of such a framework typically needs to put a larger amount of effort and time (relative to the amount needed to build a new framework from scratch) into extending the framework for his or her own target systems and experimental objectives. On the other hand, the approach of building from scratch requires the experimenter to repeat and redevelop what has already been done by earlier researchers, and gives the earlier researchers the new large burden of needing to understand and validate the new framework.

In general, repeatability of experiments is a basis of all sciences, as it makes it possible to verify the validity of experimental results. The first step that the fault injection community can make to achieve that vision would be to build a standard, extensible, and open fault injection framework. The next step would be to make an agreement on fault injection methodologies and configurations (i.e., including fault models, target system hardware configuration guidelines, standard software workloads, and experiment termination conditions). The impact would be significant. Then, an experimenter would be able to directly compare his or her experimental results to the results of others, provided that both efforts used the same fault injection configurations.

We present the *Extensible Fault Injection* (EFI) framework. EFI can emulate errors in the internal states of commodity CPU- and GPU-based computer systems (Section 4.3). The two types of fault injectors provided are:

- *Breakpoint-based fault injector*. This software-implemented fault injector can emulate errors in the OS and user processes that run on CPU-based platforms (Sections 3.3 and 4.4).
- *Mutation-based fault injector*. This source code mutation-based fault injector emulates errors in programs running on GPU devices (Section 5.8).

(ii) Measurement methods for comprehensive fault injection experiments. Fault injection shall be done in a comprehensive way. One can easily recognize the validity of that statement by considering the large size of the state spaces of modern computer hardware and software systems. Also, there is software (e.g., OS kernel or network daemons) that runs continuously. In such cases, it is unclear whether a non-manifested fault would be continuously benign (i.e., not activated or not manifested) if we monitored the software for a longer period of time. We call a fault injection experiment *temporally comprehensive* if the experiment uses a long monitoring time that is sufficient to activate as many faults as would likely appear when a target system is field-deployed. Not only that, if the variation in experimental results among different types of states or different data about the same type of states is large, a statistically significant number of fault injection samples is needed for each type of state. We call a fault injection experiment *spatially comprehensive* if we have high confidence in its results from a statistical point of view.

In order to conduct comprehensive fault injection experiments, we developed two measurement techniques. One is an *accelerated fault injection strategy* that speeds up the fault injection experiment and thus is useful for temporally comprehensive fault injection experiments within a relatively short period. Temporally comprehensive fault injection experiments are time-consuming. If a target system is monitored for a long period of time after each fault injection, the total experimentation time is significantly increased. For example, if a system is monitored for 2.5 hours after each fault injection, the experiment would take 6 to 12 months (depending on the fault type) to study just 6,000 faults. We present an accelerated fault injection strategy that can help experimenters collect a statistically significant number of fault injection samples in a relatively short time period (Section 3.4). The presented accelerated fault injection strategy reduces the experiment time by 3.2 times and 7 times for the processor and memory faults, respectively, when the monitoring time is 2.5 hours.

The other measurement technique we developed is a *data-type-aware fault injection* technique that allows us to classify fault injection results as a function of corrupted data type. Many other specific techniques have been designed that can provide either better measurement accuracy (e.g., distinguishing read vs. write activation of faults) or shorter experiment time. A spatially comprehensive fault injection experiment requires advanced fault injection methods. For example, a dynamic memory (e.g., heap) is allocated and freed at runtime. Naturally, it is difficult to know the type of data stored in a specific memory address of dynamic memory at the time of fault injection and at the time of system failure. In order to be more precise, one must be able to keep track of dynamic memory objects that are allocated and freed over time, and associate a specific data type

with a memory word inside a dynamically allocated memory object. We present a data-type-aware fault injection technique that uses a symbolic identifier to specify a fault injection target (e.g., specifying the data type of a fault injection target), converts a symbolic identifier to a virtual address at runtime, and uses the converted virtual address to set a breakpoint if the breakpoint-based injector is used (Section 3.3).

(iii) Characterization of faults via fault injection. We conducted a set of fault injection experiments on various types of COTS-based computer systems. Based on the presented validation frameworks and measurement techniques, we quantitatively evaluated the following system reliability characterization metrics (Chapters 3–4). These metrics were also modeled as mathematical formulas (e.g., curve fitting if the data follow standard distributions) for the model-based evaluation (Section 3.7).

- *Fault and error sensitivity*. We analyze the fault and error sensitivity of various types of CPU and GPU program states, where *fault sensitivity* (i.e., the non-benign faults ratio) is a ratio of the number of non-benign faults (i.e., faults that lead to system or application failure) and the total number of injected faults, and *error sensitivity* is a ratio of the number of non-benign faults and the number of activated faults.
- *Fault and error latencies*. We analyze the fault and error latencies of various types of CPU and GPU program states, where *fault latency* is the time from the introduction of a fault (fault injection) to its first activation, *error latency* is the time from the first activation of a fault to the first detection of an induced error or an induced failure, and *failure latency* is the sum of fault latency and error latency [ALR+04].
- *Error and failure location*. We analyze a distance between error location and failure location. For example, we check whether the location of an error and the location of a failure are the same program function (or software module). By checking this condition on multiple pairs of errors and failures, we compute the probability that error and induced failure are in the same function (or software module) and use this probability as a distance metric. Note that the location of a fault and the location of an error are the same if the error is one that is immediately caused by the fault and the fault is a specific type (e.g., code memory fault).

Our experiments explored the new spectrums of fault injection experiments and analyzed these metrics (see Figure 1.1). These new spectrums are long latency failures, dynamic memory faults, faults in GPU devices, and specific microarchitecture-level faults. Those four types of faults were chosen not just because they are common in commodity

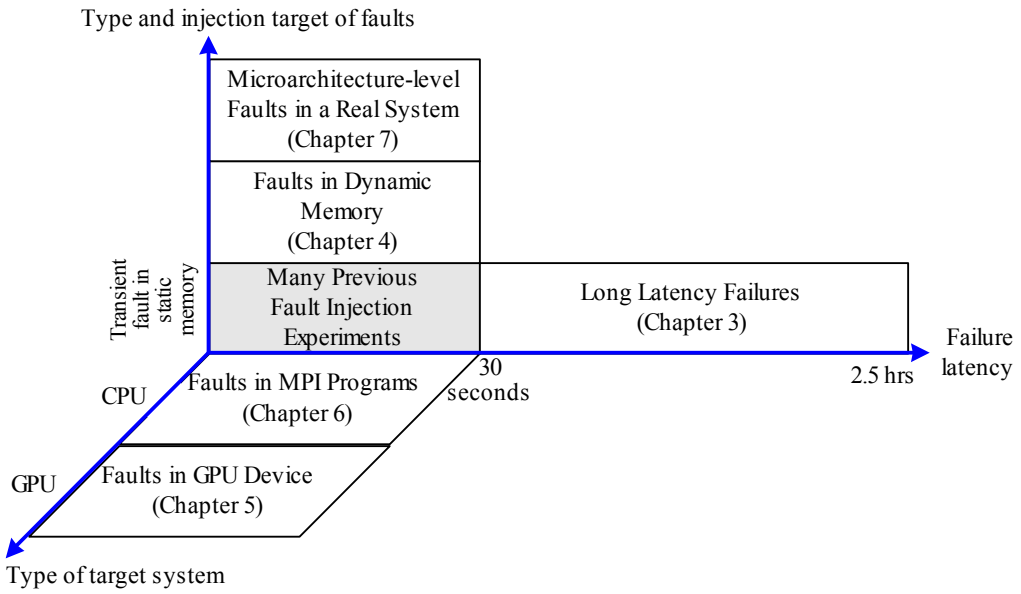


Figure 1.1. Spectrum of fault injection experiments.

computer systems, but also because they are either difficult to detect or difficult to recover from.

(a) Long latency failures (Chapter 3). We present a study on long latency failures using accelerated fault injection. The data collected from the experiments are used to analyze the significance, causes, and characteristics of long latency failures caused by soft errors in the processor and the memory. The results indicate that a non-negligible portion of soft errors in the code and data memory lead to long latency failures. The long latency failures are caused by errors with long fault activation times and errors that cause failures only under certain runtime conditions. On the other hand, because of a strong temporal locality of register values, less than 0.5% of soft errors in the processor registers used in kernel mode lead to a failure with a latency longer than a thousand seconds. Our study also shows that the obtained insight can be used to guide design and placement (in the application code and/or system) of application-specific error detectors.

(b) Dynamic memory faults (Chapter 4). We present a measurement-based analysis of the fault and error sensitivities of dynamic memory. We extend a software-implemented fault injector to support data-type-aware fault injection into dynamic memory. The results indicate that dynamic memory exhibits about 18 times greater fault sensitivity than static memory does, mainly because of the higher activation rate. Furthermore, we show that errors in a large portion of static and dynamic memory space are recoverable by simple software techniques (e.g., reloading of data from a disk). The re-

coverable data include pages filled with identical values (e.g., '0') and pages loaded from files unmodified during the computation. Consequently, the selection of targets for protection should be based on knowledge of recoverability rather than on error sensitivity alone.

(c) GPGPU program sensitivity (Chapter 5). The high performance and relatively low cost of GPU-based platforms provide an attractive alternative for general-purpose high-performance computing (HPC). However, the emerging HPC applications usually have stricter output correctness requirements than typical GPU applications (i.e., 3D graphics) do. We analyze the error resiliency of GPGPU platforms using a fault injection tool we developed for commodity GPU devices. On average, 16–33% of injected faults cause silent data corruption (SDC) errors in the HPC programs executing on the GPU. That SDC ratio is significantly higher than the one measured in CPU programs (<2.3%).

(d) Microarchitecture-level fault (Chapter 6). We find that software-implemented error detectors have time-of-check-to-time-of-use (TOCTTOU) vulnerabilities.

The experimental data have implications for the design of fault tolerance systems. A relatively small portion of faults in hardware (e.g., transistor or circuit) manifest from the viewpoint of the application software or user. For example, a fault in an unused transistor is automatically masked and does not lead to a failure. Thus, a fault tolerance technique does not have to protect all of its target system states if its cost efficiency is an important design requirement. At the same time, hardware full-duplication is conservative in the sense that many detected errors are benign for software. That consequently illustrates why error sensitivity and recoverability characteristics of target software should be considered during the design of error detection and recovery techniques.

1.3.2. Design

We present a set of fault detection frameworks and techniques for COTS-based computer systems (see Figure 1.2). The presented techniques have various distances to their protected program source code, from embedded error detectors automatically generated by a source-to-source translator to an external monitor that compares the naturally generated communication messages. By developing three types of spectrums of techniques (all of which are applicable to COTS-based systems), we are better able to understand the trade-offs among their potential benefits, limitations, and development complexities.

(i) Fault tolerance framework for GPUs (Chapter 5). We present a software framework to provide fault tolerance services for GPUs. It is a source-to-source translator that derives error detection and recovery codes for target software. The derived codes are customized and strategically placed in the target software source code following considera-

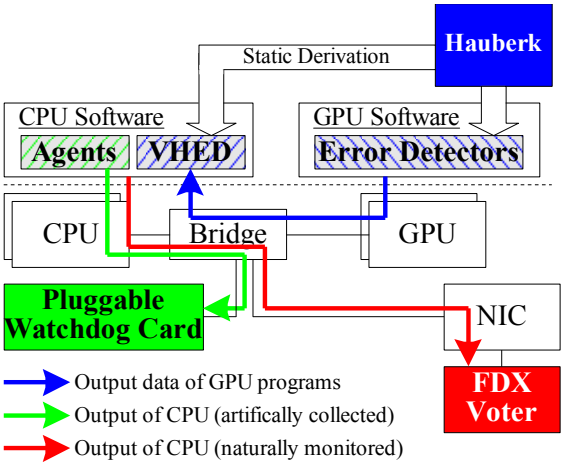


Figure 1.2. Overview of the presented fault tolerance frameworks.

tion of the common characteristics of GPU-based programs and specific characteristics of a given program (e.g., control- and data-flows). The translated codes are compiled and linked with a set of provided libraries that are called at runtime to perform fault detection and recovery operations. Using the framework, we can detect and tolerate faults in a GPU device without having to interfere with the executions of all other threads in the same program. That can significantly reduce the fault tolerance overheads in GPU-based parallel programs.

The fault detection techniques presented as a part of this framework are:

- *GPU guardian*. This technique can detect GPU hang failures and timing errors by using a timer that is set using the execution time data of the previous runs of the same GPU kernels.
- *Embedded, customized error checking*. We present embedded error-checking techniques that are embedded in the source code of the target program. The generated techniques are not only customized for the target program characteristics, but also capable of exploiting the unused parallelisms in target system hardware and the inherent memory localities of a target program. Two example techniques of this type are:
    - *Duplication-and-checksum*. This technique duplicates selected computation statements of a target program and adds checking code to compare the original and duplicated computation results in order to detect computational errors (e.g., faults in ALU/FPU). We apply this to variables that are initialized once and used as read-only for a long period of time. To protect errors that occur while the read-only data are kept in the register or

memory, we use a checksum that is XORed with the value of each protected variable twice: when the value is computed, and when the variable becomes a dead variable. We use the same checksum to protect multiple variables. Regardless of the protected variable count, the checksum shall be zero at the end of program execution. Otherwise, we detect an error in a part of registers or memory that kept the variables protected by the checksum.

- *Accumulated value range checking*. This technique is specifically designed to protect data computed inside a loop. Considering the small size of loops (i.e., as they are the main optimization targets) and the portion of total execution time taken by loops in parallel programs, the addition of a few error-checking instructions inside a loop can significantly delay the program execution time. This technique adds only two additional instructions to protect the computation done inside a loop. One instruction accumulates the value of a selected variable (i.e., one with the largest cumulative backward dataflow dependency, to which errors in other variables are most likely to propagate) over entire loop iterations. The other instruction tracks the loop iteration count. After executing the loop, each parallel thread divides the accumulated value by the loop iteration count (i.e., giving an average). The range of the average value is checked against the value ranges profiled in advance. False positives are diagnosed by the presented selective re-execution technique.

(ii) Fault tolerance framework for MPI programs (Chapter 6). We present a non-intrusive fault tolerance framework for computer nodes. The framework, named *Pluggable Watchdog*, augments the fault detection and recovery capabilities of the existing industry standard IPMI [IPMI2]. The key enhancements made in Pluggable Watchdog are the detections of hang, data omission, timing, and data corruption errors of application software. Those detections are hard to achieve unless behavioral and semantic information about application software is recognized. While Pluggable Watchdog fault detection and recovery operations run on a separate chip in the computer node level, the semantic gap between these offloaded operations and application software is addressed through use of software agents on the host computer machine.

The fault detection techniques presented as a part of the framework are:

- *Software symptom*. This technique can detect crash failures of an OS (and hypervisor) and notify the header node about them. An *OS crash* is an error or failure that is detected by the error checkers in the OS or processor/memory hardware.

Naturally, there is a software function that is called when such an error or failure detection happens. For example, specific versions of Linux go to the panic mode by calling the *panic()* function. Our software symptom-based detection adds error-reporting code in the entry of such a function. The added reporting code only uses statically allocated resources (e.g., a UDP packet) and either directly or indirectly reports this to the header node. Similarly, when a user process crashes, a specific kernel function is called that allows our technique to detect and report user crash failures as well.

- *Behavioral specification*. This technique derives a specification from the target program implementation (i.e., program binary) and enforces the specification to the program at runtime by monitoring the target program signatures generated at runtime. The specification may be, for example, a finite state automaton that captures all legal call sequences of library functions. We use that automaton to check the following four properties to detect the crash, hang, data omission, timing, and SDC failures of a target program.
    - *Termination state*. This property holds if all processes terminate at one of their final states in the specification.
    - *Transition time*. This property holds if any transition time between two states in the specification is normal. Because multiple threads execute in parallel, we can check whether the transition times of multiple threads between two identical states are similar.
    - *Output data value*. This property checks whether the values of output data produced while threads stay in the same state at the same time are similar between multiple threads in the same program.

(iii) Fault tolerance framework for COTS-based mission-critical systems (Chapter 7). In mission-critical systems, it is less important to optimize the cost of fault tolerance techniques than it is to meet the fault tolerance design requirements (e.g., "five nines" availability). The growing computational demands in mission-critical systems (e.g., a space exploration craft that must perform on-board data processing and automatic navigation) result in the use of high-performance COTS components. Compared to the previously used components (e.g., with TMR architecture), such COTS components are more likely to fail when they experience hardware faults (e.g., an alpha particle strike). The presented framework is a voter that allows developers to build highly reliable and available software-based NMR systems that use only COTS components. The presented voter is programmable and highly fault-tolerant. The fault detection techniques presented as a part of this framework are:

- *Hardware-software co-designed checking.* These techniques use both hardware and software in order to supplement the weaknesses of each. For example, all software-implemented error checkers have the time-of-check-to-time-of-use (TOCTTOU) vulnerability because of the difference between the exact execution times of checking and using operations (i.e., they are executed by different instructions). We address that vulnerability by adding an extra processor instruction that schedules a checking operation at a specific time and eventually makes the time-of-use and the time-of-check the same. We have designed some other processor instructions either to improve the coverage or to accelerate the error-checking operations implemented in software.
- *Software NMR*. This provides a programmable and fault-tolerant voter that is needed to build software-based NMR systems that use only COTS components. The large amount of redundancies created in this approach helps this technique detect almost all types of hardware faults. The presented voter makes it easy to implement complex voting algorithms and to run the voting software in a more reliable way.

(iv) Error recovery techniques. The presented detection techniques trigger error recovery techniques in order to tolerate detected errors. The following error recovery techniques are presented as a part of our frameworks.

- *Recoverability-driven memory protection* (Chapter 4). This technique reduces the protection costs under a high fault rate. Our study indicates that it is possible to recover from a significant percentage of memory errors (e.g., 70% of the static memory and 10–60% of dynamic memory errors[3]) through use of simple software techniques. For example, a large proportion of memory pages allocated by applications are filled by the same value (e.g., '0'). This technique recovers errors in these pages when the value is recorded in advance. Memory pages used for disk caches are replicated by default in stable storage. The technique recovers errors in those pages by reloading the data from the storage. Some of the user-level state can be excluded as protection targets. For example, errors in multimedia data are short-lived and quickly removed by new incoming data (e.g., the next video frame), without degrading the QoS. Our technique recovers errors in such data by

[3] In this dissertation, static and dynamic memories are defined from the viewpoint of the OS kernel. Addresses of variables in static memory are fixed at compile time (e.g., kernel global memory), while those in dynamic memory are dynamically assigned at runtime (e.g., heap object). Dynamic memory thus includes all the code, read-only data, read-write data, and stack segments of user processes and kernel modules, as well as the dynamically allocated kernel memory objects.

forcing the program to continue without any interruption. Naturally, the technique can tolerate multi-bit errors to the extent that they are detected. It can significantly reduce the error recovery overhead, because the technique is a version of forward error correction techniques that need neither a checkpoint nor a restart operation.

- *Selective re-execution* (Chapter 5). In CPU-GPU hybrid computers, in essence, we use multi-level checkpoint-and-restart techniques for error recovery. A level of checkpoint-and-restart is created for each GPU device. Another checkpoint level is created for each computer node. The other level is created for the entire system or for a part of system; in such cases, one more level is used for the entire system. Such local recovery (e.g., per GPU device) is feasible thanks to the data error detection capabilities of the presented error detection techniques. When an error alarm is raised by a GPU error detection technique, the presented selective re-execution technique re-executes the GPU kernel not only to check whether the alarm is a false positive or a true positive, but also to tolerate the detected errors without interfering with the executions of other software instances.
- *Programmable voting* (Chapter 7). This technique is intended to support software-based NMR in mission-critical systems and extremely large-scale computer systems in which failures are so common that they are likely to occur even during error recovery operations.

Chapter 2.
Background: Fault Model and Fundamental Concepts in Fault Tolerance

This chapter reviews related work that measures computer fault rates, and the basic concepts and techniques of error detection.

2.1. Fault Classification

In order to understand the problem, we classify various types of faults that exist in computer systems. No human-made artifact is free of faults. In various engineering disciplines, it is important to design a system that can avoid, detect, and tolerate failures that occur during operation. Classifying and knowing the types of faults that can occur in a target system are the first steps in the design process of fault tolerant systems. We classify faults into two major types: hardware and software faults.

(i) Hardware fault. Hardware faults are typically classified into three types based on the type of the root cause: transient, intermittent, and permanent. Both transient and intermittent faults can cause soft errors. A *soft error* is an unexpected change in the data stored in electronic circuits or transmitted between them. In contrast a *hard error* is caused by a permanent hardware fault.

(a) Transient fault. A transient fault is caused by ionizing radiation, a particle strike, or other external interference. The induced soft error can propagate and corrupt a software state, but it does not cause permanent damage to the hardware. In 1979, it was first reported that alpha particles can cause soft errors in DRAM and charge-coupled devices (CCDs) [MW+79]. An alpha particle consists of two protons and two neutrons and is produced in a radioactive decay process (namely, alpha decay). Note that ~10% of cosmic rays, which are charged subatomic particles originating from the space, are alpha particles, while ~89% of cosmic rays are simple protons. The majority of cosmic rays come from straight above the ground, according to the IBM Blue Spruce experiment [ORT+96]. In an experiment [CR+11], injecting 50 MeV protons into three COTS devic-

es caused data loss and device crash, but no symptom was observed that implied permanent damage to the devices (e.g., a high current condition).

Transient faults are usually characterized as lasting up to one clock cycle. With the scaling of manufacturing technology, transients last longer because there is less space for the particle strike energy to dissipate. In current technology, transients can last up to two clock cycles or even longer [MM10]. If scaling continues, there is a high chance this duration will become longer. Because each particle strike event is a fault, and its consequences (e.g., error and failure) are captured as the corruption of transistor output data, we classify this type of multi-cycle error as a transient fault in the device-physics layer.

(b) Intermittent fault. An *intermittent fault* has a duration that ranges between a few to billions of clock cycles [SSC03]. An intermittent fault is caused by various physical characteristics of chips (e.g., irregularities in chip voltages and temperatures). Transistors become more and more susceptible to such physical characteristics with the scaling of manufacturing technology and the integration of numerous transistors into a chip (e.g., for multi-cores and large on-chip caches).

(c) Permanent fault. A *permanent fault* occurs as a result of a manufacturing defect or the excessive use of transistors. Such excessive use degrades the reliability of transistor materials. This type of fault behaves like a long duration intermittent fault, but it permanently compromises the functionality of the transistors. Both intermittent and permanent faults are more common in high-density chips because of their high utilization and low operating voltage.

(ii) Software fault. Software fault refers to a defect or a bug in software. Software bugs can be introduced in various development stages of the software. For example, a software algorithm may have a bug. A bug can be introduced when an individual component is implemented or when multiple software components are integrated. Such faults can be activated only when a user provides one of the specific sequences of input data, which may be intentionally generated to discover common and known software faults (that are called as testing or security attack). Studies have classified the types of software faults and measured their common occurrence probabilities [RBC+92]. Those classified software faults can be emulated by translating the source code, IR code [DM06], or binary code of a target program or by changing the program state at runtime [KIT93].

In theory, a combination of hardware faults can cause exactly the same errors or failures of a software fault, however, such a combination is unlikely in nature because multiple and specific hardware faults would have to occur at the right moments. At least it is true that hardware faults are a superset of software faults. It is also true that the types of errors and failures that can be caused by software faults are similar to those that can be

caused by hardware faults. Only the exact occurrence probabilities of each type of error and failure are different. Thus, in this dissertation, we focus mainly on understanding the system behavior under various types of hardware faults, and on protecting from errors and failures caused by hardware faults. Yet the presented frameworks and techniques are also effective for many types of software faults.

2.2. Failure Rate

We survey the measured failure rates of commodity computer systems as a function of fault location. Here, we focus on failures caused by hardware faults.

In general, the system failure rate due to permanent faults follows a Weibull distribution where the rate is higher when the system is first deployed and when the system is aged than during the rest of the system lifetime. The system failure rate due to transient faults typically follows an exponential distribution, as the fault rate is constant at every moment.

Hardware faults can occur anywhere in the system. One of our target systems is a parallel computer system that uses both CPU and GPU devices. This target system has multiple types of computer nodes: *compute node* to perform computation, *header node* as an interface between users and compute nodes (to dispatch programs to compute nodes and control the launched programs via command line or GUI-based user interfaces), *storage nodes* (to provide centralized secondary storage services to the compute and header nodes), and *control node* (to monitor and manage the health of compute and storage nodes, and to provide a control interface to administrators). All four types of nodes are connected via high-speed interconnection networks.

All these nodes have all or some combination of processor, memory, and I/O devices (e.g., storage device and NIC). A compute node is the most likely to have heterogeneous computing resources. We assume that a compute node has multiple CPUs that directly interact with the I/O devices and has computational accelerators (GPUs) that accelerate the executions of CPU programs. System software on CPUs is responsible for managing I/O devices (e.g., to communicate with other compute nodes and to access the centralized file systems). GPU device drivers execute on CPUs, launch GPU kernels on GPUs by copying the program code and input data from the CPU memory to the GPU memory, and they copy the computation results back from the GPU memory to the CPU memory every time an execution of a GPU kernel is done. We assume that the memory address spaces of CPUs and GPUs are split and that the address spaces between GPUs are also

split. Data between a CPU and a GPU (and also between GPUs) is explicitly exchanged by memory copy operations (e.g., DMA or software managed I/O). A GPU device on a PCIe extension card meets these assumptions. A CPU-CPU integrated chip may also satisfy these assumptions depending on how the cache-and-memory hierarchy is organized, exposed to, and managed by software.

We survey the failure rates of aforementioned three types of hardware components where the failures are due to transient, intermittent, and permanent hardware faults as well as software faults.

(i) Processor chips. The failure rate of the ASC Q supercomputer at Los Alamos National Laboratory (LANL) with 13.88 tera FLOPS performance has been measured in 2004 [MHH+05]. This supercomputer consists of 2,408 HP Alpha Server ES45 nodes. Each node has four Alpha 21264 1.25GHz CPUs where each CPU chip is placed on a separate CPU board. In total, this ASC Q supercomputer has 9,632 CPUs. The measured failure rate of the 9,632 CPUs is 27.7 failures per week. MTTF of a CPU chip is thus ~6.669 years. This rate includes the soft error rate of CPU on-chip caches. These CPU on-chip SRAM caches are protected by a parity error checking technique but not by an ECC technique (i.e., unlike its main memory, protected by an ECC). Note that a parity technique can only detect a fault, while an ECC technique can detect and recover from a fault. The execution of all threads in the parallel program is blocked and a failure is counted whenever a data error is detected in the on-chip SRAM caches. The SRAM parity error rate is 24 failures per week. In other words, MTTF of a CPU chip due to the SRAM cache parity error is ~7.697 years.

This ASC Q supercomputer was operating at an altitude of ~2,286 meters. If the same system is hosted at sea level, where the cosmic ray induced soft error rate is 6.4 times lower than ~2,286 meters [GGR04], MTTF of a CPU chip due to the SRAM parity error would be ~49.278 years and consequently MTTF of a CPU chip would be ~25.132 years (as 3.6 failures per week are not due to SRAM parity errors). Protecting the entire on-chip SRAM caches using an ECC technique can improve the MTTF of this CPU chip up to ~51.312 years regardless of the altitude at which the machine is installed (assuming that the other 3.6 non-SRAM failures per week were not due to cosmic rays).

In a BlueGene/L (BG/L) computer at Lawrence Livermore National Laboratory (LLNL) that is larger than the ASC Q computer at LANL, all CPU on-chip caches except for the first-level (L1) caches are protected by an SEC-DED ECC technique. The L1 caches are protected by a parity technique due to their short access latency. MTTF of this BG/L computer due to the L1 parity errors is ~8 hours [GCG+07], which is the same as ~119.7 years of MTTF per CPU and is longer than ~49.278 years of MTTF of a CPU due

to SRAM parity errors of an ASC Q supercomputer machine installed at sea level. This is likely due to the smaller L1 cache size of this BG/L computer CPU than the total SRAM on-chip cache size of the ASC Q computer CPU. Still this L1 cache parity error rate seriously harms the computation efficiency of this BG/L computer because this computer in total has 131,072 CPUs (i.e., PowerPC 440 700MHz) on 65,536 nodes to provide 280.6 tera FLOPS of Linpack benchmark [DLP03] performance.

This history shows the importance of using an error recovery technique on all on-chip cache and memory devices when protecting processor chips from soft errors. Until now, many commodity CPUs, including the latest ones, are still using a parity technique but not an ECC technique to protect their L1 caches. This is mainly due to the high operational speed of the L1 caches (e.g., 1 clock cycle). Specific versions of latest GPUs for parallel computers, on the other hand, use some type of ECC techniques for their register files, all on-chip caches (including the L1 caches), and on- and off-chip interconnects. This significantly improves the resiliency of the ECC-protected GPUs to soft errors.

Regardless of the uses of ECC techniques, many subcomponents of processor chips are still left unprotected. ECC is not directly applicable to non-memory circuits (e.g., ALU and FPU). ALUs and FPUs, for example, form a large portion of the silicon area in GPUs. It is true that transient faults in these combinational circuits are less likely to manifest and propagate to flipflops and latches than those in the flipflops or latches of sequential circuits. A corrupted value successfully propagates to a flipflops or latches if and only if it arrives when the flipflop or latch is triggered[4]. The vulnerabilities of combinational circuits shall not go unnoticed, especially in large-scale parallel systems, considering the relatively large size of such circuits in high performance processors.

(ii) Memory modules. The memory fault rate is measured in many systems. Both hard and soft errors are observed. An experiment on 26 different DRAM chips using three different cell technologies shows a strong correlation between cell design and sensitivity to particle strikes, where the most sensitivity cells are 1,500 times more sensitive than the least sensitivity cells [ZNS+98]. This experiment is done by irradiation with neutrons, protons, and pions. This fundamental work partially explains the relatively large difference in the following measured soft error rates of commodity DRAM devices used in different environments.

Many early experiments report DRAM soft error rates of between 200 and 5,000 FIT (failures in time per billion operation hours) per megabits. For a computer node with 32 gigabytes of DRAM, MTTF of this DRAM subsystem due to a DRAM data error is 0.78-

[4] Latch is a level-triggered device. Flipflop is an edge-triggered device.

7.81 hours. These rates are reported by a semiconductor manufacturer (IBM) with state-of-the-art design and manufacturing technologies when this data is measured [ZCM+96][Zie96][ORT+96].

A large-scale measurement experiment of the DRAM memory error rate was conducted on six different hardware platforms between 2006 and 2008 [SPW09][SG06]. Each type of platform has several tens of thousands of nodes in production systems (i.e., Google datacenters). The reported average error rate of this experiment is 25,000-75,000 FIT per megabits, which is much higher than the previous data. Many of the observed errors are likely to be due to hard errors. Such analysis is convincing because the observed errors are strongly correlated within the same DIMM, and most of the uncorrectable errors are preceded by one or more correctable errors. Earlier work estimates that hard errors are less common than soft errors and form ~2% of all errors. Such a high contribution of hard errors in memory can be explained by considering the difference in hardware devices used by cloud servers and high availability servers (e.g., errors are more common in memory devices for non-enterprise systems). For example, a measurement that used desktop computers and commodity servers reports a relatively high hard memory error rate (4.2% = 9 out of 212 nodes) [LSH+07].

Another experiment was conducted on three distinct system environments: rack-mounted servers, desktop computers, and geographically distributed network test beds [LSH+07]. A much lower data error rate is observed here than all the previous experiments. The reported data error rate is 54.73 FIT per megabits. Although the found soft error sample count is too small (i.e., only two from 300 machines for varying multi-month periods), an interesting analysis was made that this low error rate maybe partially due to the arrangement of DIMMs perpendicular to the horizontal plane and to the small size of memory chips, as both significantly reduce the area facing the particle strikes from space.

(iii) I/O devices. Work has analyzed the disk replacement rate from a set of large production systems [SG07]. The data is from >100,000 disk drives that use three different host interfaces and are produced by at least four different manufacturers. The analyzed annual replacement rates typically exceed 0.88%, which is guaranteed in MTTF of 1,500,000 hours in the specification, with 2.4% to 3.7% common, and up to 13.6% and 24.1% on some systems. Although this replacement rate is not exactly the same as the failure rate, the common range of MTTF of analyzed disk drivers is only from 236,757 hours (for 3.7%) to 365,000 hours (for 2.4%). Similarly, network interface cards and the motherboard of the computer node can fail, but their failure rates are relatively lower than those of the other components, in general.

2.3. Basic Error Detection Techniques

This section reviews the basic concepts and techniques used in fault detection. Many fundamental fault detection techniques are designed for memory and communication channels. Parity (e.g., even or odd) is a simple technique that can detect one bit error. The concepts in the hamming code are used to design single- or multi-bit error detection techniques in a memory-space-efficient way. Hamming code has a tradeoff between the space overhead (e.g., to save the code) and the provided coverage. In order to detect and correct many more bit errors, ECC technique needs a much larger extra memory space to keep the ECC bits [BSS08]. This space overhead goes down as the protected data unit size increases.

Advanced EEC[5] organizes ECC bits in a way that the same ECC can provide higher coverage for multi-bit errors. For example, if an ECC data word is composed of bits that are physically noncontiguous, a particle strike event cannot corrupt multiple bits in the same ECC data word. Thus, an SEC-DED technique can be still used and tolerate multi-bit errors. It was measured that advanced ECC can reduce the uncorrectable error rate by a factor of 3 to 8 [SG06]. The uncorrectable error rate was from 0.25% to 0.4% per DIMM per year for SEC-DED protected platforms, and from 0.05% to 0.08% for Chipkill protected platforms.

Advanced ECC does not completely remove uncorrectable errors. This is, for example, due to accumulations of latent memory faults (e.g., two-particle strikes) [YKI09]. In such a case, multiple bits of a physically separated ECC data word can be corrupted and escape the protection provided by an SEC-DED technique.

Scrubbing addresses such latent fault problems by reducing the fault latency and removing the latent faults. The scrubbing technique periodically scans the memory data so that the used ECC technique can automatically check the integrity of data [SSP90]. Scrubbing is implemented either in hardware or software. The hardware-based implementation provides better performance than the software-based implementation (e.g., the scrubbing rate of 1 gigabyte per 45 minutes in deployed machines [SG06]).

Regarding the presented recoverability-driven memory protection, we suggested the possibility of recovering code memory using storage files for static memory [YKI09], but

[5] Advanced ECC refers to an ECC technique that can protect single- and multi-bit errors in memory by for example scattering data bits protected by an ECC word to multiple memory chips. Its commercial names include IBM Chipkill, Intel SDDC, and former Sun Microsystems Extended ECC.

the size of this area was not measured. In on-chip caches, a finding similar to ours (i.e., a large portion of zero-filled pages in memory) is reported, and this characteristic is used to design a cost-efficient compressed cache [YZG00]. The low error sensitivity of visual data is reported [NPB08], but the memory size of these data in an integrated system is not measured.

Error detection is similar to outlier or novelty detection. The existing outlier detection techniques are generally classified into four types.

(i) Distribution-based. This approach uses the statistics theory. It [BL94] tries to best fit the given data samples to a standard distribution function (e.g., normal or Poisson). Discordancy tests are used to detect outlier samples where the tests support both univariate functions and some multivariate functions (e.g., normal [But83]). For example, a discordancy test declares a sample as an outlier if it lies ≥3 standard deviations from the mean assuming that the data follow a normal distribution [FPP78]. This type of technique assumes the underlying distribution of sample data is known and is similar to a standard distribution. However, distributions of the runtime signature data of computer software are not always known and not all signature data may follow standard distributions when mixed workloads run together.

(ii) Depth-based. This approach [Tuk77] maps each data sample to a point in a *k*-dimensional (*k*-d) space and computes the depth of each point. A point with the smallest depth has the highest likelihood as an outlier. One simple definition takes the minimum of the number of samples to the left of a sample *x*, and the number of samples to the right of a sample *x* where $k = 1$ (i.e., one-dimension sample space) [RR96]. Many different depth definitions exist that show high detection accuracy. The computation of depth relies on the computation of *k-d* convex hulls (i.e., smallest volume in the *k-d* space that contains all sample points and straight lines between any pair of sample points) that has a lower time complexity bound of $\Omega(n^{\lceil k/2 \rceil})$ for *n* samples, making this approach impracticable for $k > 4$ and large *n*.

(iii) Distance-based. This approach declares a sample *x* as an outlier if at least *p* percent of samples lie at a greater distance than *D* from *x*. If the total population is *P*, this condition is the same as where the distance between *x* and its *k*-nearest neighbor (kNN) [RRK00] is more than *D* where *k* is $1–p/P$. This simplified condition can provide similar detection coverage if *D* is chosen properly. For example, the exact value of *D* can be computed to check samples that lie ≥3 standard deviations from the mean, assuming the samples follow a normal distribution. Simple algorithms with the time complexity $O(kN^2)$ exist, where *k* is the dimension of a sample space, and *N* is the sample count. A nested-loop algorithm compares the distance of each pair of samples (e.g., total N^2 pairs) and stops this

process for sample x when the samples with the distance less than D to x are more than p percent. This simple algorithm can be tuned in a way to maximize the memory locality. A cell-based algorithm [KN98] with the time complexity of $O(c^k+N)$ has good scalability for large data (i.e., large sample data is kept in the storage) by forming nearby samples as cells and increasing the memory locality. Also, there is a technique that profiles the normal behaviors of a system (e.g., system call sequences [FHS+96]).

(iv) Classification-based. Anomaly-based detection is widely used in intrusion detection system (IDS) to detect malware and intruders in computer nodes and networks [FHS+96][LSC97][LSM99]. These techniques monitor a well-selected set of system-level software events (e.g., system call) and hardware events (hardware performance counters). These techniques then use various types of classification techniques to train the classification networks in an efficient way (e.g., k-nearest neighbor, local outlier factor [AMA+11], and Bayesian networks for software faults). Many of these existing techniques use offline training because of the large amount of computing power needed to perform such operations and the difficulty of online diagnosis of security attacks. These techniques are implemented as a part of the software of the monitored system (e.g., OS kernel or hypervisor [ALL06]) or of an external device.

Chapter 3.
Long-Latency Failures: Measurement Technique and Characterization

This chapter presents a measurement-based analysis of the significance, causes, and characteristics of long-latency software failures caused by soft errors in the processor and memory. The analysis is possible because we developed a software-implemented fault injection framework to accurately control the fault injection target, to collect the specific failure information (e.g., failure latency, location, and type), and to accelerate the fault injection experiments. The results indicate that a non-negligible portion of soft errors in the code and data memory lead to long-latency failures. The long-latency failures are, for example, caused by errors with long fault activation times and errors that cause failures only under certain runtime conditions. On the other hand, less than 0.5% of soft errors in the processor registers used in kernel mode lead to a failure with a latency longer than a thousand seconds, mainly because of a strong temporal locality of the register values. Based on the results, we present a new classification of failures using the knees in fault and error latency distributions to provide useful insight into where to focus the protection mechanisms so as to provide good coverage while reducing overhead. Moreover, we conducted a curve-fitting study to model the failure latency distributions of failures caused by four types of faults.[6]

3.1. Motivation

Soft errors in the processor and memory present a critical problem for the dependability of system software running on large-scale or mission-critical computer systems. A s*oft error* is an unexpected change to data stored in an electronic circuit or transmitted between circuits. Its main causes include physical stimuli and variations (e.g., a particle strike, cosmic ray, voltage/thermal variations, and silicon wear-out). Soft errors are a

[6]Part of this chapter was published: K. S. Yim, Z. Kalbarczyk, and R. K. Iyer, "Quantitative Analysis of Long Latency Failures in System Software," in *Proceedings of the Pacific-Rim International Conference on Dependable Computing*, pp. 23-30, 2009.

paramount concern in the design and operation of supercomputers, large-scale cloud datacenters, and aerospace computer systems. For example, the MTTF of a processor of the ASC Q supercomputers in Los Alamos National Laboratory was 6.7 years in 2004, and 86.7% of the failures were due to soft errors in the parity-protected on-chip caches [MHH+05]. ECC protection significantly improves the MTTF of a processor (e.g., a supercomputer that has ECC-protected on-chip caches [GCG+07]). Note that the number of processors used in state-of-the-art supercomputers grew by a factor of 103 from 2004 to 2011, making the overall system-level processor failure rate worse. Memory devices also are not free from soft errors. The measured failure rate of DRAM in cloud data centers was 25,000–75,000 FIT per billion operation hours per megabit in the years between 2006 and 2008 [SPW09], where failures were due to both transient and permanent hardware faults. The failure rate has increased mainly because the process technology of processors and memory has reached the deep nanometer scale, and multiple dies and cores, as well as large-size caches and memories, are integrated into chips.

System software (e.g., operating system) failures due to soft errors harm the availability of all application programs executing on the system. System software in high-performance servers and mission-critical, embedded computers runs continuously without restarting (e.g., for a month). That long execution time can affect error propagation characteristics, because the longer execution increases the chance of activating a latent fault.

Not many previous fault injection experiments, however, have monitored the error propagation behavior for a long period of time after injecting a fault. As far as we know, with a modern PC one of the longest monitoring periods was 30 seconds, and it was done by a software-implemented injector [GKI04]. In almost all simulated fault injectors, the monitoring time was shorter than several milliseconds because of the slowness of simulation when highly accurate models were used. There was the technical difficulty of collecting a statistically significant number of long-latency failure samples. For example, if a target system were monitored for 2.5 hours after each fault injection, the fault injection experiment would take at most 12,500 hours (i.e., about 1.4 years) to examine 5,000 faults, where only a small fraction of the examined faults would lead to long-latency failures. (See Section 3.4 for accurate evaluation models and data.)

In previous fault injection experiments, it has repeatedly been found that most soft errors cause short-latency failures. In both x86 and PPC machines, over 95% of non-benign soft errors in the processor registers, stack, and code and data memory caused system failures within 200 milliseconds (corresponding to an execution of about 1 billion instructions on a 3 GHz processor with 1.5 instructions per cycle) from the time the inject-

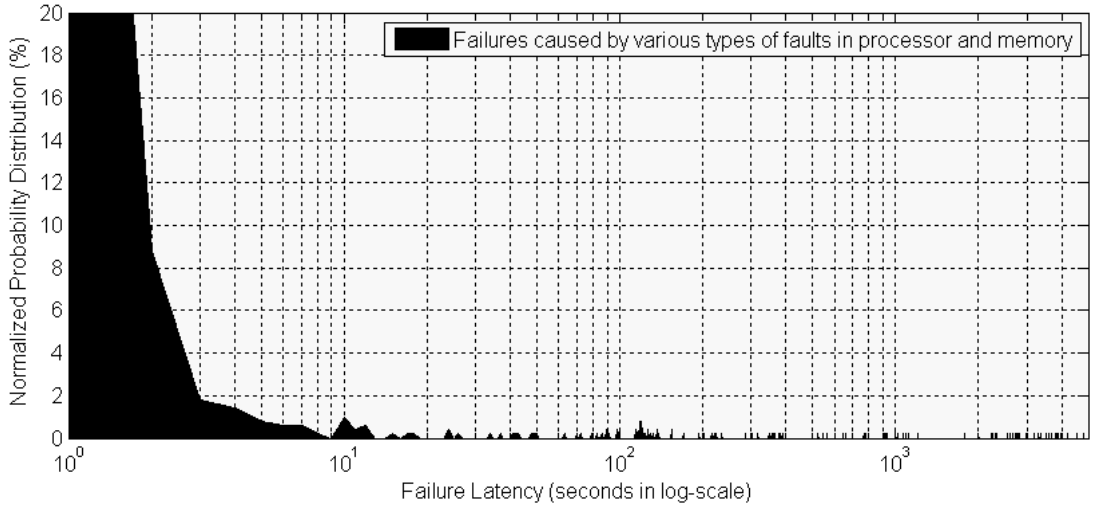


Figure 3.1. Probability distribution of failure latency.

ed fault was activated [GKI04]. Even shorter failure latency was found with hard errors in the processor. In processor registers, 86% of permanent errors that occurred when applications were being run and 99% of those that occurred when the OS kernel was being run were detected in <100 thousand instructions [LRS+08]. Similar findings were reported by several earlier studies. For example, in a machine widely used in the early 1980s, 70% of errors were detected on the day the fault was injected, and 12% on the following day [CI87]. Since the machine processed ~500 thousand instructions per second, one day in that early experiment is equivalent to 10 seconds on a 3 GHz modern processor processing 1.5 instructions per cycle.

In the previous work (as discussed above), failure latency is measured after fault activation, and thus fault latency is not taken into account. Note that *fault latency* is the time from the introduction of a fault (fault injection) to its first activation; *error latency* is the time from the first activation of a fault to the first detection of an induced error or an induced failure; and *failure latency* is the sum of fault latency and error latency.

In this study, long latency failure is defined as a failure whose latency is longer than the knee of a failure latency distribution of similar faults. Figure 3.1 shows the normalized probability distribution of failure latency obtained by a preliminary study in which various types of soft errors were injected into the Linux kernel. Here, a majority of failures had latencies shorter than 2 seconds, equivalent to the execution of about 10 billion instructions in the machine we used. Also, a non-negligible portion of failures had latencies longer than 2 seconds and formed the long tail in the probability distribution function. These long-latency failures are the target of our analysis.

This chapter presents a measurement-based analysis of the significance, causes, and characteristics of long-latency software failures caused by soft errors in the processor and memory. The analysis is possible because we developed a software-implemented fault injection framework to accurately control the fault injection target, to collect the specific failure information (e.g., failure latency, location, and type), and to accelerate the fault injection experiments.

The main contributions of this study can be summarized as follows.

- We developed a fault injection framework to accurately measure the occurrence times and locations of fault, error, and failure events (Section 3.3).
- We designed an accelerated fault injection strategy, and optimized and evaluated the total fault injection experiment time of this strategy (Section 3.4).
- We analyzed the long-latency failures in quantitative and qualitative ways using the monitoring time of 2.5 hours (Sections 3.6 and 3.8.1).
- We present a new classification of failures using the knees in fault and error latency distributions to provide useful insight into where to focus the protection mechanisms so as to provide good coverage while reducing overhead (Sections 3.6 and 3.8.2).
- We conducted a curve-fitting study to model the failure latency distributions of failures caused by four types of faults (Section 3.7).

3.2. Related Work

This section reviews the related fault injection tools, experimental methods, and experiment results.

3.2.1. Fault Injection Tools

The existing fault injection frameworks can be classified into three basic types:

(i) Simulated fault injectors [KIR+99][KIT93][WQR+04]. Simulated fault injectors use software models of hardware devices. They are essential in studying the impact of faults in the device-physics layer (e.g., simulating an energetic particle strike on transistors). Thanks to the improved simulation power, this type of injectors is now used to simulate processor subsystems, memory subsystems, and even whole computer nodes. A simulated fault injector changes a state in a simulated model in order to emulate an error induced by a hardware fault, while executing software on top of the model.

A system-wide simulated fault injector has to deal with a trade-off between modeling accuracy and simulation speed. An accelerated simulator chooses the simulation speed

(e.g., using a dynamic binary translation) but at the same time degrades the simulation accuracy. For example, in our preliminary study, we found multiple cases in which an accelerated simulator (i.e., QEMU[7]) crashed even before the simulated software failed. Many other simulated fault injectors use highly accurate models that can cover both normal and abnormal execution behaviors. Conversely, high-accuracy models slow down the simulation speed (e.g., to hundreds kilo instructions per second, KIPS). Thus, system-wide simulated fault injectors are often used to monitor system behaviors up to several hundred milliseconds or several seconds after each fault injection. Such simulation time can be sufficient if the main purpose of the experiment is to analyze the impacts of faults in a hardware subcomponent (e.g., at the transistor, logic, or microarchitecture level).

(ii) Hardware-implemented fault injectors [MRM+94][BGG+02][AKT+08]. Hardware-implemented fault injectors either use an external tool or modify the design of a target hardware device in order to emulate an error in a target system state. If an external tool is used, accurate control of fault injection time and location is difficult because the clocks of the tool and target system need to be synchronized. Thus, in practice, those methods allow an experimenter to inject a random fault into a specific signal but may not allow it when a specific type of data is transmitted on the signal bus (e.g., if the signal bus transfers various types of data and has a high clock speed).

Embedding a fault injection tool as a part of the design of a target hardware device addresses that control accuracy problem. For example, a method could rewrite the HDL code of a target hardware device. The drawbacks would be that it requires the source code of the target hardware device and may incur large initial development costs for the hardware design modification (e.g., testing and fabrication). That method is thus used for systems built from programmable hardware (e.g., FPGA).

(iii) Software-implemented fault injectors (SWIFI) [SFB+00][AFR02][BCS+90][CMS98][KIT93][KKA95][KA95]. A number of SWIFI tools have been deployed to support the characterization of software behaviors in the presence of hardware and software faults. The use of SWIFI can speed up the fault injection experiments (as the target software runs at a normal speed on top of real hardware), provide highly accurate measurement data, and reduce the experiment cost (as no hardware modification is necessary). Thus, SWIFI is now the most widely used among the three common types of fault injection tools. State-of-the-art SWIFI tools use a hardware breakpoint feature in order to

[7] QEMU (Quick EMUlator) is an open source computer emulator that uses a dynamic binary translation technique. This shows a high emulation speed because it executes the translated code on top of the native host machine and caches translated binary codes for future reuses. As a result, QEMU is also used as a virtual machine monitor (VMM) or as a part of VMM. QEMU is available at http://qemu.org.

emulate an error when a specific state of the target software is being accessed. That breakpoint-based fault injection does not require any modification in target hardware and software (so it is applicable to commodity systems) and has a small performance overhead, at least insofar as it is supported by a target processor (which is the case in many modern processors). On the other hand, this breakpoint-based injection can introduce a large performance overhead for particular types of hardware faults (e.g., intermittent or permanent faults), as breakpoints need to be repeatedly set.

3.2.2. Failure Latency

The data in the previous fault injection experiments showed that a large portion of transient faults in modern computers have short failure latencies. That finding inspired the design of modern dependable computer systems. For example, Chillarege and Iyer [CI87] reported that 70%, 82%, and 91% of memory faults were detected the same day, by the following day, and by the third day, respectively. Because the used machine processed about 500 KIPS, one day in this early experiment is equivalent to about 10 seconds on current-generation hardware. That early work also showed a correlation between the failure latency of memory faults and system utilization (e.g., higher utilization resulted in shorter failure latency). Another experiment monitored its target system for a long period of time after a fault injection (e.g., 20 minutes) by using software-implemented fault injectors. The experiment showed that failure latencies of memory and software faults are usually long (e.g., >20 minutes), while those of bus and CPU faults are short [KIT93]. Shorter failure latency was found with hard errors in the processor. 99% of failures due to processor hard errors that occurred inside an OS kernel were detected in <100,000 instructions when the monitoring time was 10 million instructions [LRS+08].

In the early 1980s, while much work focused on measuring or estimating failure latency, Shin and Lee [SL86] measured fault and error latencies separately for the first time. Because an injected fault is dejected after a time t, if there is a failure, it can be determined whether the fault latency of the injected fault is shorter than t. Repeating that experiment multiple times with different values of t gives an estimate of fault latency distribution. In Shin and Lee's experiment, fault latency was measured up to a few seconds. Our experiment also separately monitors the fault and error latencies of faults in various types of data by using a method that directly measures the fault and error latencies of each fault.

2.1.3. Fault and Failure Accelerations

Fault injection experiments are an accelerated way to test system reliability because they *accelerate the fault occurrences*. By emulating faults or errors at a rate much higher than the rate at which they appear in nature (e.g., using an ion accelerator), a fault injection experiment can evaluate the reliability of a chip in a short time period. Using the fault injection data, one can build hardware reliability models that evaluate the reliability of a design before the chip is fabricated and released (i.e., before field measurement is possible) [ZMM+96].

Failure acceleration is a technique to reduce the fault injection experiment time. According to a definition by Chillarege and Bowen [CB89], the maximum failure acceleration is realized when the fault latency is zero, the error latency is a minimum, and the fault sensitivity (i.e., the probability of a fault is causing a failure) is a maximum. For example, a fault can be injected into a hot page (i.e., with a strong memory locality) to reduce the fault latency because the injected fault is likely to be activated in the near future. It is possible to reduce the error latency by running a workload under a high-utilization condition that in fact accelerates both the fault activation and the error propagation. By injecting severe faults (e.g., multi-bit errors), one can also increase the fault sensitivity.

That earlier failure acceleration concept changes the time domain or the distribution of different types of faults (that is as compared to those in nature) to accelerate failures. Thus, it is not suitable for failure latency measurements (i.e., measuring fault and error latencies). In this chapter, we present a fault injection strategy that reduces the fault injection experiment time by increasing the fault activation ratio, while maintaining the ability to collect error and failure information accurately.

3.3. Tool: Fault Injection Framework

The presented framework is designed to accurately measure fault, error, and failure latencies as well as the error and failure locations.

- *Fault and error latencies*. We analyze the fault and error latencies of various types of CPU program states.
- *Error and failure location*. We analyze the distance between error location and failure location. For example, we check whether the location of an error and the location of a failure are the same program function (or software module). By checking this condition on multiple pairs of errors and failures, we compute the probability that the error and the induced failure are in the same function (or soft-

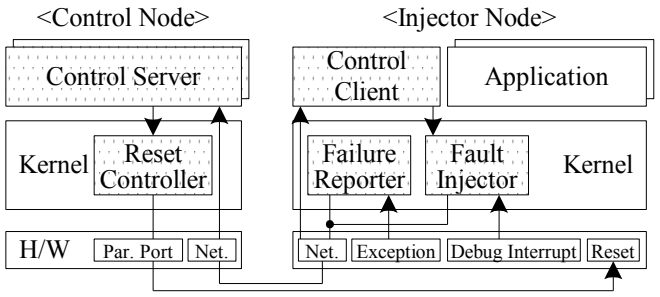


Figure 3.2. Control architecture of the presented fault injection framework.

ware module), and use this probability as a distance metric. Note that the location of a fault and the location of an error are the same if the error is one that is immediately caused by the fault and the fault is of a specific type (e.g., a code memory fault).

Figure 3.2 depicts the system and software architecture of our FI framework. It consists of two types of nodes: control and injector nodes. A *control node* provides an interface between the user and the injector nodes. A fault injection experiment is conducted with an *injector node* that also reports the experiment results to the control node.

The control node runs a GUI-based control server that allows the user to specify a fault injection campaign, execute the campaign on a set of injector nodes, and collect the injection results. All these operations are performed in cooperation with the control client running on the injector node. The control client generates a heartbeat signal to the server to notify the server about the health of the injector node. Based on the control command sent from the server (via a TCP/IP channel), the client configures its fault injector module and executes the requested benchmark applications.

The *fault injector* is a kernel module residing on the injector node and is capable of injecting faults into various parts of the system: the code and data memory, the stack, and the general- and special-purpose processor registers in the kernel and the user address spaces.

A log reporter residing in the injector node is responsible for reporting the kernel crash information to the control server. The log reporter collects information about the exception type (raised upon a node crash) and program location that caused the failure. The failure location is identified through analysis of the call stack of the thread that caused the exception. The log reporter then sends (as a UDP packet) the collected information to the server.

The *control server* detects other types of failures by monitoring the messages sent from the control client. A kernel hang is detected through monitoring of the heartbeat message. If a UDP packet containing crash information does not arrive, the server infers that the heartbeat stopped because of a kernel hang rather than a kernel crash. When the benchmark application in the injector node completes, the control client sends the application output to the server. A prematurely reported output indicates an application crash. No output reported indicates an application hang. Incorrect output data imply silent data corruption.

The injector node is rebooted after each activated fault, i.e., when the corrupted instruction/data is executed/used. The control server triggers the reset signal of the injector node by using the reset controller, a device driver of the parallel port control cards installed in the control node. The parallel port signals are connected to the hardware reset signals of all injector nodes, and thus the reset controller can generate a reset signal for a specific injector node.

3.3.1. Collecting Error and Failure Information

Our FI framework is designed to accurately collect fault, error, and failure information. That information is useful, for example, in studying the error propagation paths of a target system. Technically, the control node (or control nodes that relay messages between the master control node and a set of injector nodes) collects the type, time, and location information of the error and the first failure caused by an injected fault. Note that information on an injected fault does not need to be collected, because it is already owned by the control node. In the rest of this subsection, we describe how we used the breakpoint-based injector on an x86 ISA as an example to explain the error and failure information collection methods.

(i) Error information. The location of an error is extracted from a fault injection command sent from the control server. The time and type of an error are collected through use of the fault activation packet. That activation packet (e.g., using UDP) is sent by an injector node to its control node when the injection target is accessed for the first time (see Figure 3.3). If the breakpoint-based injection mode is used, the packet is sent by the breakpoint handler. The breakpoint handler reads the bit-code of the current instruction of a preempted process (or thread) and sends this bit-code and the processor register context of the process to its control server. The *disassembler* library in the control server is used to analyze the activation type: read activation or overwritten. Overwritten refers to a case when the corrupted value is overwritten by a new value before the first use (i.e., a

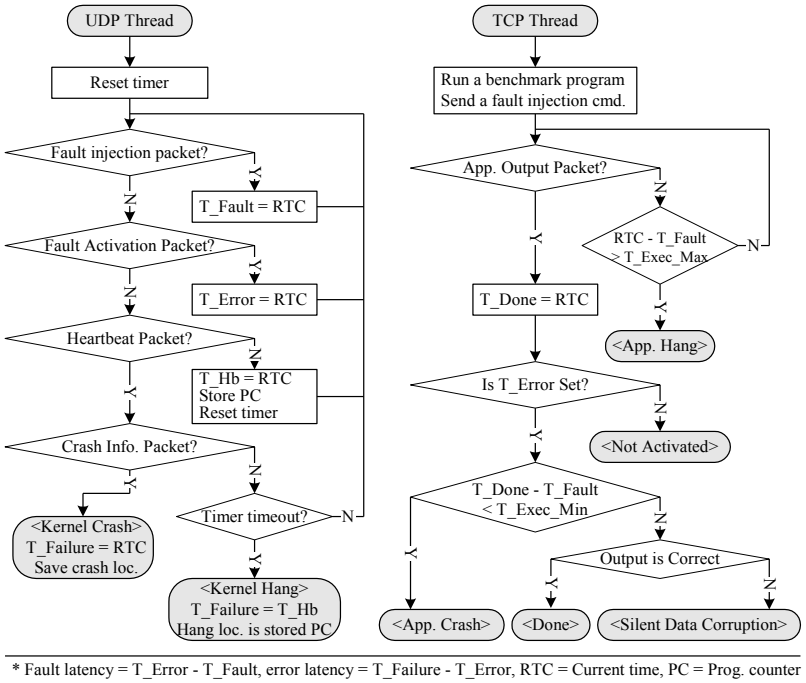


Figure 3.3. An algorithm to collect error and failure information.

type of error masking as the corrupted variable is being dead in a program at the time when the fault is activated).

The activation time can be measured by two methods. One uses the local system clock of the control node. It is useful if the fault latency is significantly longer than the tit takes to transmit an activation UDP packet from an injector node to a control node. Fault latency is computed through subtraction of the fault injection time (i.e., the time when a fault injection command was sent by a control server) from the fault activation time (i.e., the time when the fault activation packet arrived at the control server). The other method uses performance profiling hardware available in modern processors. Performance profiling hardware typically gives the number of spent cycles or executed instructions between two time points (e.g., at fault injection and activation). This cycle count (or instruction count) is sent to a control server to calculate the fault latency. The same methods are used to measure the error latency, which uses the fault activation time and failure time (e.g., the time when the first crash information packet arrived at the control server from an injector node, see Figure 3.3).

(ii) Failure information. The failure information is collected by two threads on a control node (see Figure 3.3). These UDP and TCP threads process UDP and TCP packets,

respectively, sent by each injector node. Four components in the breakpoint-based injector node can send these packets. They are: the control client, kernel watchdog, user heartbeat generator, and failure reporter.

(a) OS crash. The type and location information for OS crash failures are derived as follows. When the OS of an injector node crashes[8], the call to an exception handler is intercepted by the *failure reporter*. The failure reporter collects the type of the invoked exception handler (i.e., crash failure type), the PC register value of the crashed process (i.e., crash failure location), and the call stack information of the crashed process. The crash type and the PC value are directly read from the process control block (PCB) and kernel stack of the crashed process. That collected information is used to compose a crash information UDP packet that is sent to a corresponding control server. The call stack information is obtained through tracking of function frame pointers stored in the kernel or user stack of the crashed process. For example, in x86 ISA, the frame pointer (i.e., *ebp*) register can store the pointer to a memory word that contains the frame pointer register value of the caller function depending on the enabled compiler option. The address of the call instruction of the caller is obtained through reading of the memory value right above the location where the value of frame pointer register of the caller is saved. That read PC (i.e., *eip*) register value is used to find the symbol name of the caller function. The process is repeated recursively until the bottom of the call stack is reached. That derived call stack information is useful for the post-mortem analysis of error propagation paths.

(b) OS hang. The location of OS kernel hangs is not easy to measure. The first technical challenge is that of gaining control from a hanging OS kernel. We configured the advanced programmable interrupt controller (APIC) so that it could periodically issue a non-maskable interrupt (NMI). The NMI handler sends the number of timer interrupts that have occurred during the last NMI interval and the PC register value of a preempted process to its control server as a payload of a UDP heartbeat packet. The control server detects an OS kernel hang if no heartbeat packet is seen for a certain time interval (e.g., 5 seconds) or the timer interrupt count accumulated for a certain time interval is zero. The hang location is identified through manual source code analyses using the collected PC values.

[8] For the explanatory purposes, we assume that target system directly runs the OS on top of its bare hardware. This however does not mean that the presented techniques are inapplicable to systems that use system-level (type-I) and/or user-level (type-II) hypervisors. In fact, many of them are orthogonal to the use of hypervisors (e.g., if applied to each guest OS). While parallel computers (e.g., many supercomputers) typically do not use hypervisors to maximize the performance, hypervisor is a standard in distributed systems (e.g., clouds). We thus discuss the applicability of the presented frameworks and techniques to hypervisor environments whenever it seems non-trivial.

(c) User process failure. The TCP thread (see Figure 3.3) is used to detect user process failures. The control client in an injector node sends the output data of a benchmark program to its control server when the program finishes. If the output data packet does not arrive after the maximum expected execution time of the program, this is treated as a user process hang. If the output packet arrives but the execution time is shorter than the minimum expected execution time, we treat this as a user process crash. If the output data are incorrect (i.e., different from the golden output), a user process SDC error is declared.

Note that when a fault causes a kernel crash or hang, the control node resets the injector node. If the injector node does not reboot properly after the reset (e.g., after two tries), the most likely cause is a file system corruption that could not be fixed by a file system integrity checker (e.g., *fsck* in UNIX and Linux). That would happen if the injected error was propagated to file system metadata or system file data. In that scenario, it would be possible to install the OS and all necessary programs of the injector remotely from the control node using the pre-boot execution environment (PXE). The failure of a remote installation implies at least one hard error in the injector node.

3.3.2. Parallel Fault Injection

Parallel fault injection uses multiple nodes for the concurrent execution of a fault injection campaign. The cost efficiency of that approach is represented by the ratio of the number of injector nodes to the number of control nodes. We achieved the ratio of 8 to 1 with commodity hardware. Multiple control servers ran on a single machine, and each server responded to different TCP and UDP ports. A master control server generated a list of fault injection commands according to the user-provided fault injection campaign and stored the list to a file. All control servers accessed that file and dequeued a command for execution. The access was protected by a lock. As a result, we achieved a linear speedup directly proportional to the number of injector nodes. Note that we abstracted the heterogeneity of the injector nodes by using a symbol name to specify a fault injection target, because different machines can use different virtual addresses, even for the same injection target.

3.4. Measurement Method

The divide-and-conquer fault injection strategy is designed to quickly identify benign faults and non-benign faults that have short failure latencies. The strategy then studies the remaining faults one by one, and hence can reduce the experiment time while maintain-

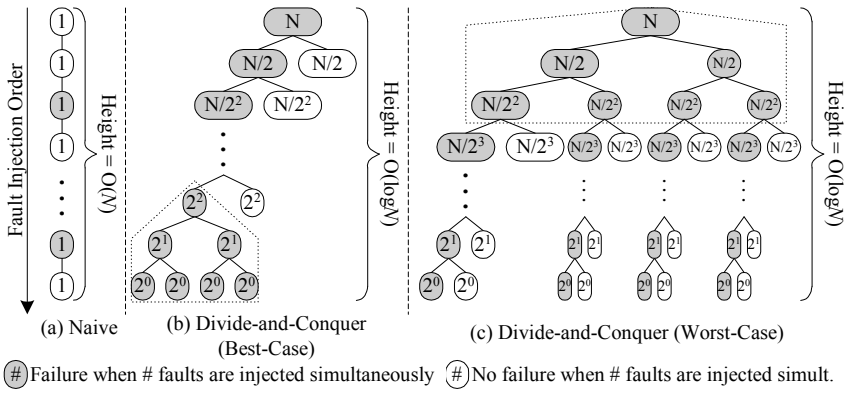


Figure 3.4. Fault injection strategies (f=4).

ing the ability to accurately measure the type, latency, and location information for all injected non-benign faults.

3.4.1. Fault Injection Strategy

A conventional fault injection strategy repeats the same fault injection process N times to study N fault samples (see Figure 3.4(a)). The repeated process injects a fault, monitors the behavior of a target system (e.g., up to a certain monitoring time), and restarts a target system (e.g., using the *reset controller* in the control node, see Figure 3.2).

In the presented divide-and-conquer strategy, all N faults are injected one-by-one, and a short monitoring time (e.g., 60 seconds) is used to find non-benign faults that have short failure latencies[9]. The unidentified faults (e.g., neither manifested nor activated during the 60 seconds) are passed to the second phase. In the second phase, all passed faults ($\overline{N}$) are injected simultaneously (see Figure 3.4(b) and 3.4(c)). If a failure is observed, we divide $\overline{N}$ faults into S subsets and perform a fault injection experiment for each subset. If no failure is observed in a subset, we treat all the injected faults as benign, although some of them were just not activated faults during the short monitoring time. Conversely, if a subset causes a failure, it is once again divided into S subsets where each set has about $\overline{N}/S^2$ faults. The procedure is repeated until the subset size becomes 1. At that point, since only one fault is injected, we can accurately collect error and failure information (e.g., latency, type, and location).

The presented fault injection strategy is implemented in the following two phases:

[9] Failure latency is a time from the occurrence of a fault to the detection of an induced error or failure. This is the sum of fault and error latencies.

Table 3.1. Algorithm of the divide-and-conquer fault injection strategy.

```
01: function early_manifest(fault, result, n)
02:   n_ll = 0
03:   for i = 1…n
04:   | inject_one(fault[i], result[i], T_short);
05:   | if result[i].failure is empty
06:   | | fault_ll[n_ll] := fault[i]
07:   | | result_ll[n_ll] := empty
08:   | | n_ll := n_ll + 1
09:   div_n_conq(fault_ll, result_ll, n_ll);

10: function div_n_conq(fault, result, n)
11:   if n is 1
12:   |   return inject_one(fault[0], result[0], T_long);
13:   inject_all(fault, result_all, n, T_long);
14:   if result_all.failure is not empty
15:   | pivot := n/2;
16:   | div_n_conq(fault[1…pivot], result[1…pivot], pivot);
17:   | div_n_conq(fault[(pivot+1)…n], result[(pivot+1)…n], n-pivot);
18:   else
19:   | for i = 1…n
20:   | | result[i] := result_all;
```

(i) Early manifestation phase. Many failures have relatively short failure latencies in modern computer systems (e.g., [YKI09]). This phase quickly studies such failures. In this phase, all N faults are injected one-by-one by using the *inject_one* function (see line 4 in Table 3.1) with a short monitoring time (T_{short}, e.g., 30 seconds). In Table 3.1, the *"fault"* variable is an array that contains fault injection commands, and "*result"* is an array that contains the fault injection results. We empirically set the short monitoring time value to a time where a majority of non-benign faults can manifest (see Section 3.4.2).

If an injected fault does not manifest quickly (line 5), either the fault is benign, or it has a long failure latency. Its fault injection command is stored (lines 6-8) and passed to the next divide-and-conquer phase (line 9). The number of faults passed to the next phase is $\overline{N}$ (i.e., *n_ll* in Table 3.1). $\overline{N}$ is less than or equal to N.

(ii) Divide-and-conquer phase. This phase is to efficiently search the non-benign faults among the passed $\overline{N}$ faults. In this phase, we simultaneously inject $\overline{N}$ faults, and monitor the system for the long monitoring time (T_{long}, line 13). If a failure is observed (line 14), we divide the $\overline{N}$ faults into S subsets to perform a fault injection experiment on each subset (lines 15-17). If no failure is observed in a subset (line 18), we assume that all faults in this subset are benign (lines 19-20). If a subset shows a failure, all faults in the subset are recursively divided into another S subsets where each set has about $\overline{N}/S^2$ faults (S is 2 in Figure 3.4 and Table 3.1). This procedure is repeated until the number of

faults in the subset becomes 1 (line 11). When the subset size is 1, because only one fault is injected (line 12), the error and failure information for this fault is accurately measured.

This divide-and-conquer phase requires simultaneous injection of multiple faults. In practice, there is a limit on the number of hardware breakpoints that can be armed at the same time (e.g., up to 4–8). Thus, software breakpoints are used. For processor register faults, software breakpoints are set in all target instructions to obtain the control of a system before each of the target instructions is executed. For memory faults, multiple faults are preemptively injected into the memory space just after a benchmark program is started. However, that prevents the experimenter from measuring the activation information of benign faults (e.g., faults excluded in any experiment that has a fault subset size of >1). That is the only known drawback of the presented divide-and-conquer fault injection strategy.

3.4.2. Evaluation

We compare three strategies: naïve, simple divide-and-conquer (i.e., using only the second phase of the presented strategy), and the presented strategy (i.e., using both the first and second phases). We used a trace-driven simulation with real failure latency data in order to evaluate the experiment time. We used this analytical approach because it can reduce the evaluation time relative to an alternative approach that directly measures the experiment times of all three evaluated strategies by using the same set of faults.

(i) Baseline model. Figure 3.4 illustrates an example fault injection campaign in which 4 faults cause failures (i.e., $f = 4$) and S is 2. Each node in the graph depicts a single fault injection experiment, and the number specified in the node is the number of faults injected simultaneously. *White circles* correspond to nodes with no detected failures, and *gray circles* represent nodes with at least one detected failure. The best case is when all non-benign faults are in a subset whose size is $\lfloor f/S + 0.\dot{9} \rfloor S$ as shown in Figure 3b). Similarly, as shown in Figure 3.4(c), the worst-case is when all non-benign faults are uniformly distributed across multiple subsets. The number of required fault injection experiments is given by (3.1).

$$\left(\sum_{i=0}^{\log_S f} S^i\right) + C(\log_S N - \log_S f) \tag{3.1}$$

The left term represents the sub-tree indicated by a dotted pentagon in Figure 3.4(b) and 3.4(c) and calculates the number of nodes in a perfect binary tree with a height of $\log_S f$. The right term corresponds to the rest of the tree where ($\log_S N - \log_S f$) is its height and C is the number of nodes at each level of the tree. Thus, C is S in the best case, and C is Sf in the worst case.

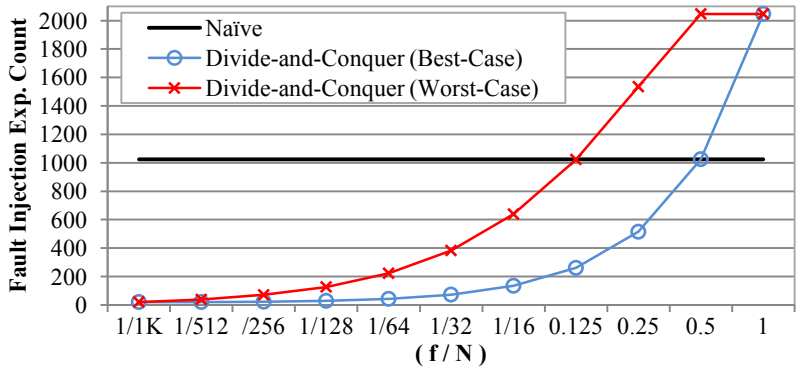


Figure 3.5. Experiment count of naïve vs. simple divide-and-conquer strategy.

Figure 3.5 shows the numbers of fault injection experiments needed by the naïve and the simple divide-and–conquer strategies for N equals 1024. The speedup ratio is determined by two parameters: the ratio and distribution of non-benign errors. The simple divide-and-conquer strategy reduces the fault injection count if the non-benign error ratio is less than 12.5% (see Figure 3.5). For an error ratio of 1.5%, the 23x and 5x speedups (relative to the naïve strategy) are expected in the best case and the worst case, respectively. Note that the worst case corresponds to a uniform distribution of non-benign errors. In practice, a uniform distribution of non-benign errors is not typically observed, because data of the same types are usually stored in close proximity and are likely to have similar fault sensitivities (i.e., at least, similar error sensitivities).

(ii) Extended model. We then developed an extended model to evaluate the total experiment time as a function of the used fault injection strategies and the type of injected faults. We built extended analytical models for the three strategies. The models accurately calculate the fault injection experiment time by using the failure latency distribution data as input. The failure latency data are collected through the fault injection experiments (described in Section 3.3.1). The failure latency data came from about 6,000 failure samples. We collected the samples by injecting faults into four types of program states: code memory, data memory, control register, and data register.

(a) Naïve strategy. The total experiment time of the naïve strategy ($TT_{naive}(S_{1...N})$) is given by (3.2).

$$TT_{naive}(S_{1...N}) = N \cdot T_{setup} + \sum_{i=1}^{N} L_{fail}(S_i) \quad (3.2)$$

Here, N is the total number of examined faults. S_i is the i-th fault injection experiment. T_{setup} is the experiment setup time, which was set to 90 seconds in our evaluation. $L_{fail}(S_i)$ is the failure latency of S_i. $L_{fail}(S_i)$ is the same as the average execution time of a benchmark program if the fault S_i is neither activated nor manifested, and is the same as the

maximum execution time of the benchmark if the fault causes an application hang. The monitoring time (T_{mon}) is used as the maximum execution time of the benchmark program.

(b) Simple divide-and-conquer strategy. The total experiment time of the simple divide-and-conquer strategy ($TT_{dnc}(S_{1\ldots N})$) is given by (3.3).

$$TT_{dnc}(S_{1\ldots N}) = T_{setup} + \min_{i=1\ldots N} L_{fail}(S_i) + TT_{dnc}\left(S_{1\ldots\lfloor N/2\rfloor}\right) + TT_{dnc}\left(S_{\lfloor N/2\rfloor+1\ldots N}\right) \quad (3.3)$$

We assume that the failure latency of an experiment (that simultaneously injects multiple faults) is the same as the minimum failure latency of experiments for which each experiment injects only one fault. Equation (3.3) is applicable if at least one injected fault is non-benign. If all injected faults are benign, the total experiment time is given by $TT_{dnc}(S_{1\ldots N}) = T_{setup} + \min_{i=1\ldots N} L_{fail}(s_i)$.

(c) Presented strategy. The total experimental time of the presented strategy ($TT_{pres}(S_{1\ldots N})$) is given by (3.4).

$$TT_{pres}(S_{1\ldots N}) = N \cdot T_{setup} + N \int_{t=0}^{T_{mon;short}} t \cdot P(t)\, dt + \bar{N} \cdot T_{mon;short} + TT_{dnc}(\bar{S}_{1\ldots\bar{N}}) \quad (3.4)$$

Here, $\bar{N}$ is the number of faults that did not manifest in the first phase where $\bar{N}$ can be calculated by (2.5). *P(t)* is the probability of having a failure at time *t* when all N faults are injected. *P*(*t*) is derived from the measured failure latencies (L_{fail}) of the used failure samples. $T_{mon;short}$ is the monitoring time of the early manifestation phase, and $\bar{S}_{1\ldots\bar{N}}$ is a set of faults with a failure latency longer than $T_{mon;short}$. In (3.4), the integral term is the experiment time of the early manifestation phase for the faults with a failure latency shorter than $T_{mon;short}$ and the $\bar{N} \cdot T_{mon;short}$ term is for the rest of the faults. The TT_{dnc} term is for the experiment time of the divide-and-conquer phase.

$$\bar{N} = N\left\{1 - \int_{t=0}^{T_{mon;short}} P(t)\, dt\right\} \quad (3.5)$$

The three models were used with the measured failure latency data. Figure 3.6 shows the total experiment times for the three strategies, normalized to a case in which 1,000 faults are studied.

The experiment time was inversely proportional to the fault sensitivity and directly proportional to the failure latency of the injected faults. For example, in the naïve strategy, the experiment time for the control register was shorter than that for the general-purpose register because the control register faults had a higher fault sensitivity. In the control register and code memory, the fault sensitivities were similar, but the control register had a shorter experiment time than the code memory because most non-benign control register faults had a short failure latency (e.g., <5 seconds). Because of those characteristics,

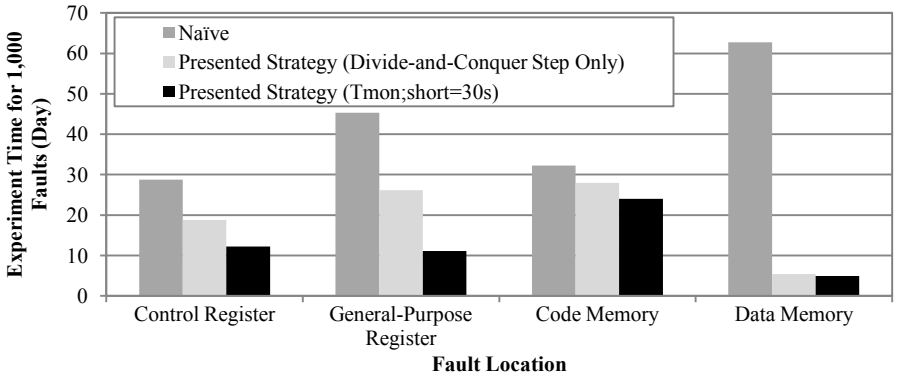


Figure 3.6. Experiment time vs. Fault injection strategy (Simulation).

data memory faults required a long experiment time (e.g., ~63 days with the naïve strategy).

The simple divide-and-conquer strategy reduced the experiment time for control register, general-purpose register, code memory, and data memory faults by 35%, 42%, 13%, and 91%, respectively, relative to the naïve strategy. The reduction ratio is larger if many of the examined faults have a shorter failure latency (e.g., register faults) and the fault sensitivity is lower (e.g., data memory). If failure latency is short, the divide-and-conquer phase progresses quickly to small-size fault subsets. A small subset is likely to have no non-benign faults. Similarly, if fault sensitivity is low, the divide-and-conquer phase has a higher chance of having a subset without any non-benign faults.

The two-phase divide-and-conquer strategy further reduced the experiment time over the simple divide-and-conquer strategy. When $T_{mon;short}$ was 30 seconds, the two-phase strategy reduced the experiment time by 34%, 58%, 14%, and 8% for control register, general-purpose register, code memory, and data memory faults, respectively, relative to the simple strategy. A large reduction was achieved when the failure latency was short (e.g., register faults). The reason is that faults with short failure latencies are identified in the first phase and excluded in the following examinations.

Figure 3.7 shows an optimization space of the presented strategy. Here, we controlled the $T_{mon;short}$ parameter. There were two suboptimal points for each fault type. The first suboptimal point was when $T_{mon;short}$ was 30–60 seconds. That point optimized the early manifestation phase. The second suboptimal point was when $T_{mon;short}$ was 120–240 seconds. That point optimized the divide-and-conquer phase. Note that the y-axis of Figure 3.7 is a log scale. Based on the optimization results, we generally recommend that 30 seconds be used as $T_{mon;short}$.

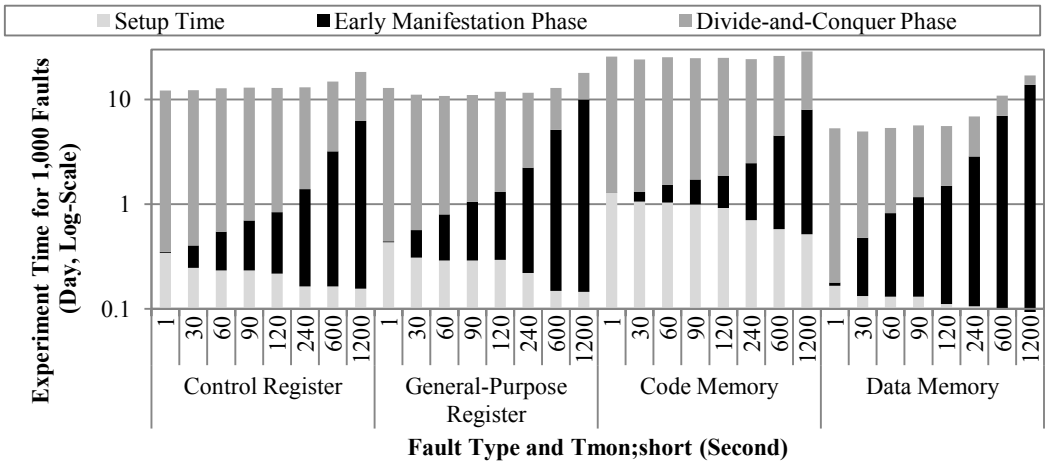


Figure 3.7. Optimization results of a parameter in the presented strategy.

In summary, the presented two-phase strategy is a generic form of the naïve and the simple divide-and-conquer strategies. Specifically, if T_{short} of the presented strategy is 0, then it works the same as the simple divide-and-conquer strategy. If $T_{short} = T_{long}$, the presented strategy works the same as the naïve strategy because the second step is not needed (i.e., $\overline{N}$=0). On average, the presented two-phase strategy improved our experiment speed by 3.2 times and 7 times for processor register and memory faults, respectively, relative to the naïve strategy, when the maximum monitoring time (T_{long}) was 2.5 hours. If the monitoring time is longer, the presented strategy can provide a higher experiment time speedup ratio thanks to the larger reduction in the experiment times for benign faults and non-benign faults with short failure latencies.

3.5. Experimental Setup

We have conducted a series of fault injection experiments on commodity computer systems. The data analyzed in Section 3.6 were obtained using the following fault injection methodology.

For software on CPUs, we used the presented fault injector to emulate a hardware fault that has successfully propagated to a software-visible architecture state (e.g., processor register or memory data). The majority of our fault injection targets were selected from OS kernel states because of the importance of the OS reliability (because the reliability of all user processes depends on it). We executed a benchmark program in the user space, while performing a fault injection experiment in the OS kernel space. In order to

examine various workloads we used a widely accepted kernel stress test suite (Linux Test Project, LTP[10]). The execution time of LTP was about 1.5 hours in the used injector nodes. We monitored each injector node up to 2.5 hours in order to classify the differences between the performance variations of a target system and hang failures.

(a) Memory faults. Memory faults were injected into static text segments of the OS kernel, dynamic text segments of the kernel modules, and static data segments of the kernel (namely *rodata*, *data*, and *bss*). We injected ~4,000 faults into those segments. The total sizes of the static kernel text and data segments were ~1.8 MB and ~1 MB, respectively. Both single- and double-bit errors were used as fault models. We assumed that the probability of memory transient faults (e.g., soft errors) has a uniform distribution across the entire physical memory space. Fault injection targets were thus uniformly selected from the entire target memory space. For kernel code memory, we randomly selected 135 kernel functions from all kernel functions, and injected faults into randomly chosen instructions of these 135 kernel functions. For kernel static data memory, faults were uniformly injected into the entire memory space of the selected data segment.

(b) Processor faults. Register faults were injected into some of the most frequently executed kernel functions. We chose the functions by profiling the OS kernel (i.e., using OProfile[11]). We injected about 2,000 faults (specifically, single-bit errors) into the general-purpose data (e.g., *eax*, *ebx*, *ecx*, *edx*, *edi*, and *ebp* in x86 ISA), control (e.g., *esp*, *eip*, *eflags*, the return address stored in the system stack, and the instruction register in x86), and special-purpose registers (e.g., *cr2*, *fs*, and *gs* in x86), while running the 100 most frequently used kernel functions. The 100 functions represented ~78% of the total execution time of the kernel if the LTP benchmark executed in user spaces. We used only the activated faults (e.g., a breakpoint is triggered) in our analyses. Note that we selected targets using execution frequency information in order to accurately reflect the probability of a transient fault in the processor registers. That is a frequently executing function is more likely to experience a processor transient fault than a less frequently executing function is.

3.6. Result

By analyzing the fault injection experiment result data, we characterize the fault and error latencies, and the distance between fault and failure locations.

[10] The LTP Benchmark, http://ltp.sf.net.
[11] OProfile is a system-wide performance profiler for Linux and is hosted at http://oprofile.sf.net.

Table 3.2. Fault Sensitivity vs. Monitoring Time

Fault Location		Monitoring Time (Minute)				
		1	10	30	60	≥90
Memory	Code	17.5%	23.3%	25%	26.3%	27.1%
	Static Data	1.4%	1.6%	2.2%	2.7%	3%
General-Purpose Register	Control	75%	76.7%	76.7%	76.7%	76.7%
	Data	20.1%	20.1%	20.1%	20.4%	20.4%

* Target system is Linux v2.6.16 on x86 processors. This experiment is done in 2009.

We present a new classification of failures based on fault and error latencies. It classifies failures into SFSE, LFSE, SFLE, and LFLE failures, where "S" means short, "L" means long, "F" means fault latency, and "E" means error latency. The threshold between short and long latencies is defined through consideration of the knee of the latency distribution and the cost of recovering from the failure. In general, it is difficult to recover from long-latency failures with a checkpoint technique. In particular, long-error-latency failures (e.g., SFLE) are more problematic than long-fault-latency failures (e.g., LFSE). The reason is that long fault latencies are due to latent faults whose presence does not affect the execution behavior of a system (e.g., program output).

3.6.1. Fault and Error Latencies

We analyze the fault and error latencies. Our analysis shows that thanks to the short error latencies it is possible to recover from many failures via a backward error correction technique or other simple software techniques.

(i) Long Monitoring Time. Previous work found that a large portion of masked errors may be due to the use of a monitoring time that is too short to allow for activation of all injected faults (e.g., 30 seconds to 1 minute). In order to determine whether a masked fault is permanently masked or just unactivated under the used benchmark workload, we conducted another fault injection experiment using a longer monitoring time (2.5 hours) and executing a stress test case suite for the Linux OS (LTP).

Table 3.2 shows the fault sensitivities (i.e., percentages of OS kernel failures) as a function of the monitoring time and the fault location.

Observation 3.1. *After sufficient time to examine the OS kernel states (e.g., >90 minutes) had passed, the fault sensitivities were at least 9.6%, 1.6%, 1.7%, and 0.3% higher for code memory, data memory, control register, and data register faults, respectively, than those measured when the monitoring time was 1 minute.*

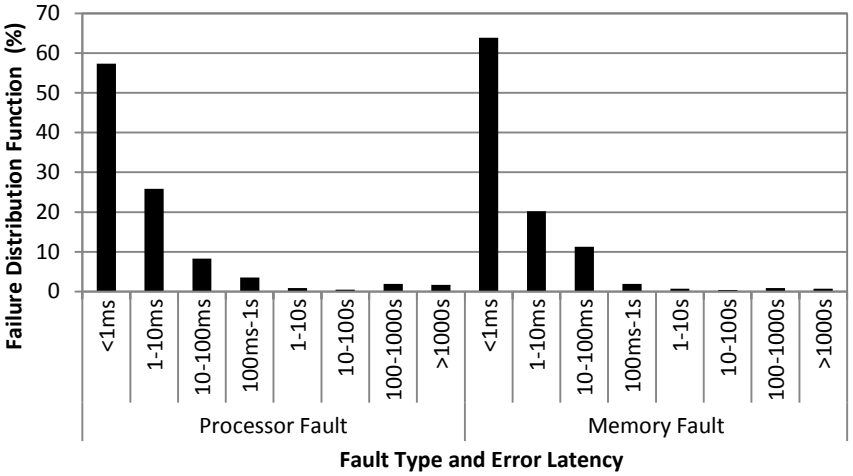


Figure 3.8. Error latency distribution of processor faults.

In particular, register faults had smaller differences than memory faults. For memory faults, as the monitoring time became longer, the increase in the rate of fault sensitivity decreased. Note that the used LTP benchmark is a *de facto* standard stress test suite for the Linux OS that examines a large portion of the practically accessible kernel code and data.

(ii) Error Latency. Figure 3.8 shows the error latency distributions of failures caused by memory and processor faults. The types of data in which faults were injected were the code and data memory segments, and the general-purpose control and data registers. Figure 3.8 shows that a majority of failures have short error latencies if faults occur in one of those four locations.

Observation 3.2. *~95% and ~97% of failures due to memory and processor faults, respectively, had error latencies shorter than one second.*

The phenomenon is due in part to the strong temporal localities of accessed values, the high clock speed of modern processors, and the use of various types of error detectors in modern computer systems. For example, if corrupted data are accessed, those data are likely to be accessed many times in a short time interval. If the corrupted data have a high error sensitivity, during the short time interval, there is a high chance of causing a failure. If there is no failure during the interval, it would indicate that the corrupted data have a low error sensitivity; thus, the error is likely to benign and is unlikely to cause a failure at its next accesses (i.e., the error has a long error latency).

Those data imply that one of the error propagation paths of long-error-latency faults is through the memory state. For example, if the address operand of a memory write in-

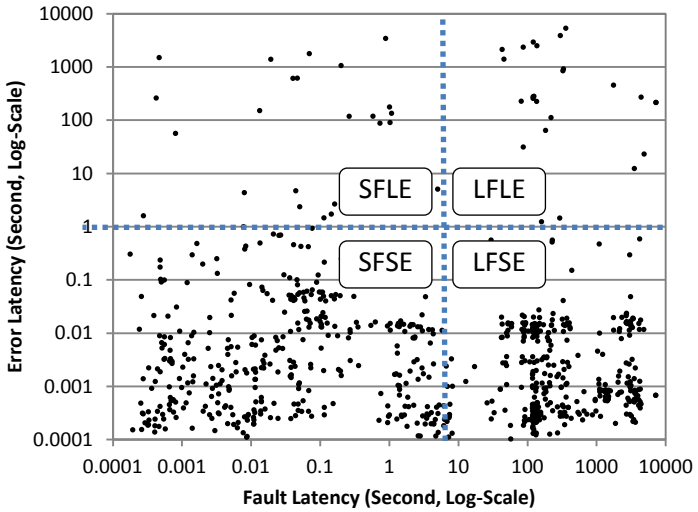


Figure 3.9. Fault and error latencies of memory failures.

struction is corrupted, it can corrupt an arbitrary address data in the memory. It may take a long time before the corrupted data are accessed (i.e., the corrupted address escapes the memory locality of the host program. We describe more detailed analyses of such error propagation paths in Section 3.8.

(iii) Fault Latency. Figure 3.9 plots the fault and error latencies of failures caused by non-benign memory faults. The dotted lines in the figure classify these failures into the four types: SFSE, LFSE, SFLE, and LFLE. The data in Figure 3.9 show a long tail in the fault latency distribution (e.g., many LFSE type failures) but not in the error latency distribution (e.g., only a few SFLF-type failures).

Observation 3.3. *47% of non-benign memory faults have fault latencies longer than 10 seconds.*

Fault latency of memory faults depends on the temporal locality of corrupted memory data, which is determined by the characteristics of workloads (e.g., regularity of system calls in OSes and user inputs in applications). We found a strong correlation between the fault latency and the access frequency of data into which a fault has been injected (i.e., temporal locality). The average fault latency depended on the execution frequency of a function into which faults were injected. Even in the same function, fault latency of a fault in the code memory varied depending on the execution frequency of a basic block containing the faulty instruction. Because of those correlations, the workload characteristics directly impact the fault latency.

Table 3.3. Distance between error and failure locations

Error Latency	Fault Type	Error and Failure Locations	
		Same SW Module	Different SW Module
Short [GKI04]	Processor	90%	10%
	Memory	92.5%	7.5%
Long >10 sec	Processor	25%	75%
	Memory	54.5%	45.5%

3.6.2. Fault and Failure Locations

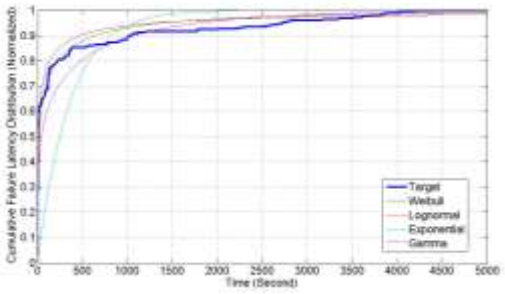
Table 3.3 shows the percentage of failures detected in a software module in which faults have been activated (or where errors have occurred). Failures with long error latencies (i.e., > 10 seconds) were more likely than those with short error latencies to be seen in a different software module. *Specifically, 90-92.5% of short-error-latency failures were detected in the same OS module in which a fault was injected [GKI04]. On the other hand, 25–54.5% of long-error-latency failures were detected in the same OS module as the fault injection.* The reason was that long-latency failures were not quickly detected and were likely to propagate and corrupt system states that were outside the software module where the error occurred.

3.7. Modeling

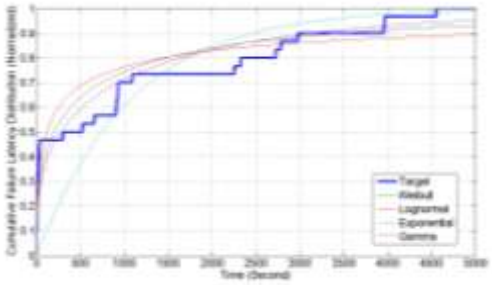
Statistical data that characterize system failure behaviors offer a basis for modeling fault-tolerant computer systems and analyzing system dependability and performance metrics under various operational conditions. For example, such an analytical failure model can be directly used as part of a system model and can also be used to derive formulas that directly compute certain metrics.

We present analytical parameter models that statistically capture system failure behaviors. Specifically, we give an example of how the measured failure latency can be fitted into known probability distribution functions.

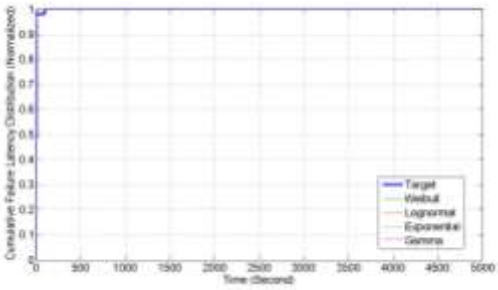
In Figure 3.10, the thick blue line is the failure latency distribution of four different types of faults studied in the experiments described and analyzed in Section 3.6. Let us assume that $L_{failure}(t)$ is the probability of having a failure latency t for a fault of the considered type. Specifically, failure latency t means that the fault latency is x and the error latency is $(t - x)$ where $0 \leq x \leq t$. This is formulated in (3.6).



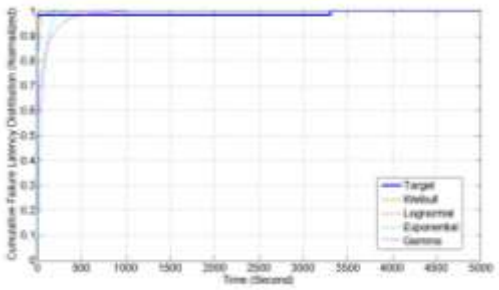
(a) Code memory fault.



(b) Data memory fault.



(c) Control register fault.



(d) General-purpose data register fault.

Figure 3.10. Curve fitting example of failure latency distribution.

$$L(t) = \int_0^t L_{fault}(x) L_{error}(t-x)dx \tag{3.6}$$

Here, $L_{fault}(t)$ and is the fault latency distribution and $L_{error}(\mathrm{t})$ is the error latency distribution. Those distributions were computed by using the latency distributions of faults leading failures (i.e., non-benign faults and detected benign faults), because the rest of the faults were not detected and thus neither have an error latency nor a failure latency. Because of such normalization, $\int_0^\infty L_{failure}(t)dt = 1$.

We fitted the four known probability distribution functions to the $L_{failure}(t)$ of code memory, data memory, control register, and general-purpose data register faults. Figure 3.10 shows examples in which Weibull, lognormal, exponential, and gamma distributions have been fitted into the measured failure latency distributions. Here, failure latency distribution is used for explanatory purposes. In practice, both fault and error latency distributions can also be used (e.g., as part of model-based studies).

3.8. Characterization

This section analyzes the error propagation paths of long latency failures and their implications for error detection.

3.8.1. Error Propagation Paths

We analyze the propagation paths of undetectable errors. Many detected errors change the control data of a program (e.g., control flow or memory address), while many undetected errors change only the non-control-data of a program. These undetected long-latency errors are classified into two types:

(i) Long-error-latency crashes. Failures have long error latencies if their error propagation paths are unlikely to be (or are not easily) checked by the baseline error detectors. We classified long-error-latency failures into two types: crash and hang.

The analyzed causes of long-error-latency crashes include the corruption of error-insensitive data and the propagation of corrupted data to memory data that have weak access locality. A fault in a processor register can propagate to another register when the corrupted register is used as an operand of an arithmetic, register-to-register move, branch, call, or return instruction (see Figure 3.11. for an error propagation model). The baseline error detectors in a processor can monitor the operands of some of these instructions (e.g., the target address of a call or return instruction). When an undetected error is used as an operand of a memory store or a stack push instruction, it can propagate to memory (see Figure 3.11(ii)). Depending on the locality of the corrupted memory data, it may be a long time before the corrupted data are re-accessed (e.g., fetched to a processor register by a memory load or a stack pop instruction, see Figure 3.11(iii)). Even if the corrupted memory data are loaded into a register, a failure is caused only when the corrupted data are used for an error sensitive operation in which the operand integrity is checked by the baseline error detectors. For example, if there are multiple code fragments that can read the corrupted memory data, the failure latency is long if only a small number of the code fragments can execute the types of operations that are sensitive to the corruptions in their operand data.

Following analysis of the long-error-latency crash failure samples obtained in our experiment, we classify the long-error-latency crash failures into the following three types:

(a) Errors propagating from register to memory. We found that when a register variable that was going to be stored in memory was corrupted, it was likely to cause a failure with a long error latency. In the case of data pushed to the stack, the pushed data are typically popped to a register at the function exit. The corrupted source register in the processor is overwritten by another value just after executing the push instruction. Thus, a failure can occur only when the restored value is used for at least one error-sensitive opera-

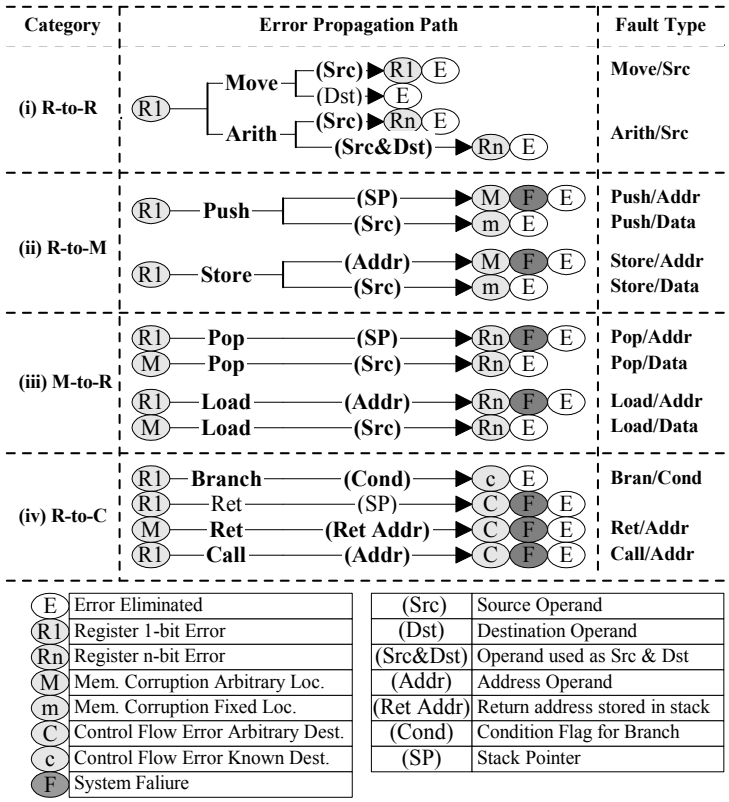


Figure 3.11. A model of error propagation paths.

tion. Because the corrupted variable can have a live range similar to the function length (which includes all the functions called by that function), the error latency can also be as long as the function execution time.

(b) Errors escaping memory access locality. Memory access locality plays an important role in the error latency. Typically, data stored in register, cache, and memory are accessed again in the near future, because of the temporal locality. However, when the address of a memory store operation is corrupted, this stores the value to an arbitrary memory location. If the data originally stored in this arbitrary location have a weak memory locality, it is likely to take a long time to access the corrupted data stored in the memory (e.g., the corrupted data escape the locality of the original memory data), and thus a long-error-latency failure can occur.

(c) Needing multiple activations. The above two cases showed that many long-latency failures propagate through the memory regardless of the original location of the fault. If the resulting corruptions in specific types of memory data (e.g., due to memory faults) manifest only under certain runtime conditions (e.g., depending on the caller func-

tion), it can take a long time to use the corrupted data, and thus a long-error-latency failure is likely. For example, if there are five functions that can use the corrupted data, and one of them can lead to a failure but its execution frequency is lower than that of the other four callers, it may be that the fault in the data can be activated quickly when one of the five functions is called, but that the induced error manifests only when that one specific function is executed (i.e., the one with a long error latency). It is also true in faults in the code memory. When the second *je* (jump equal) conditional branch instruction of the *find_pid()* function of the Linux OS on an x86 computer is changed to *jo* (jump overflow), this function can return an incorrect *pid* structure instance. This corrupted *pid* instance can be used by five functions, breaking the data integrity of a *task_struct* instance. In our experiment, a failure was detected when *flush_old_exec()* called *de_thread(task_struct *tsk)* and the *de_thread()* function checked whether *tsk->signal->count* is 1.

(ii) Long latency hangs. Our fault injection results showed that it is important to detect unconventional OS kernel hangs. In our experiments, a NMI-based preemptive hang detection was used. Kernel hangs were immediately detectable if the interrupt service was stopped. We found that not all kernel hangs disabled the vital OS services (e.g., the interrupt or processor scheduler) in a short time interval. Such hangs took a long time to detect with the hang detector we used.

We describe two of the long latency hang cases we found:

(a) Gradual error propagation. One type is due to errors that propagate over a long period of time. The propagated errors gradually disable different parts of the OS kernel services. The error is detected when the propagated errors harm the availability of the critical system services (e.g., the scheduler) or states that are checked by the used error detectors.

(b) Manifestation under certain conditions. Another type occurs when an error propagates and stays in memory (similar to a long-latency crash). For example, in our fault injection experiment, the OS kernel stuck at a while loop, *while(slabp->inuse < cachep->num && batchcount--)* defined in *cache_alloc_refill()* of the slab allocator of the Linux OS. When that hang occurred, *(slabp->inuse < cachep->num)* was false, and *batchcount--* was not executed. Thus, *batchcount* did not change and the loop condition of the outer while loop, *while(batchcount > 0)*, was never unsatisfied when *batchcount* was higher than 0. In the normal case, when *(slabp->inuse < cachep->num)* was false, the condition of an if-statement *(slabp->free == BUFCTL_END)* inside the outer loop was true, because all slab objects were in use, and no more free objects existed in the given slab. The execution of the *if-then* paragraph eventually made the loop condition of the outer loop false. This hang was thus caused by the broken integrity of the memory data, *slabp-*

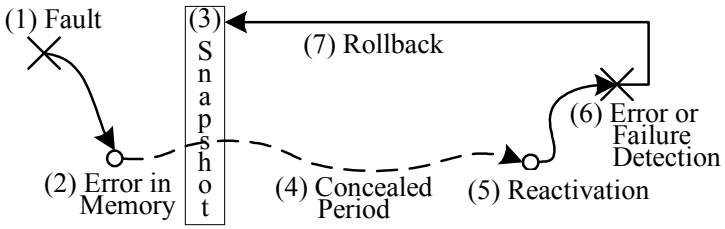


Figure 3.12. Long latency failure vs. checkpointing.

>*inuse* and *slabp->free*, and this hang had a long error latency because of the time gap between the error (i.e., the corruption of the *inuse* or *free* variable) and hang failure.

3.8.2. Implications for Error Detection

We eventually detected many long-latency failures, but it was difficult to recover from them. From the error recovery point of a view, we classify long latency failures into two types. The following recoverability analysis of those types shows the importance of early and accurate detection of such long latency failures.

(a) Long error latency. One type is long-latency failures that have a long error latency (e.g., *SFLE* or *LFLE*). In this case, checkpointing techniques suffer from the large overheads due to the checkpoint corruption problem. That problem increases not only the restart and rework times of a checkpoint-and-restart technique but also the checkpoint storage size (to keep multiple checkpoints). Figure 3.12 illustrates a checkpoint corruption problem in which the fault, (1), can be either a soft error in memory or a fault in the processor that propagates to the memory. A checkpoint (or program snapshot), (3), made after a memory error, (2), can be corrupted and hence should not be used to correct the error. To guarantee a successful recovery, the application should roll back to a checkpoint (not shown in Figure 3.12) taken before the occurrence of the memory error (2). That increases the space overhead required to store checkpoints for recovery. Unlike parallel programs, many distributed programs (e.g., interactive programs) have very short checkpoint intervals (e.g., less than a second). In such cases, one should be cautious when employing a checkpointing technique to recover from runtime errors, especially if the memory is not well-protected; we found that most long-latency failures are caused by errors in memory rather than in the processor registers.

(b) Long fault latency. The other type is long latency failures due solely to long fault latency (e.g., *LFSE*). This type of failures is easier to recover from, because the incorrect output data are producible only after the fault activation. There is no problem associated with the I/O recovery. Moreover, if an incremental checkpointing technique is used, as

long as the page that contains such a latent memory fault has not been modified since the last checkpoint save, the corruption does not propagate to the checkpoint storage. Such latent faults can be corrected through reloading of all pages at the time of the rollback (or restart) operation. If a non-incremental checkpointing technique is used, the checkpoint corruption problem still exists in this type of long latency failures.

3.9. Summary

We have investigated the significance, causes, and characteristics of long latency failures caused by transient hardware faults in processor and memory devices. 47% of non-benign memory faults have fault latencies longer than ten seconds. Advanced ECC and memory scrubbing techniques can remove such latent memory faults if the overhead of such techniques is allowed in the application domain. Still, ~5% and ~3% of failures due to memory and processor faults, respectively, have error latencies longer than one second. Among them, the processor faults are neither detectable nor tolerable by the advanced ECC and memory-scrubbing techniques. The majority of the found long-latency failures are due to corrupted memory data that have been unused for a long period of time. This shows the importance of *early detection of memory data corruption errors* especially if high availability is required, because it is difficult to recover from long-latency failures by for example using a checkpoint-restart technique.

Chapter 4.
Data-Type-Aware Fault Injection on Multiple Computer Systems

This chapter presents a measurement-based analysis of the fault and error sensitivities of multiple computer systems. We develop a software-implemented fault injector to support data-type-aware fault injection into three different types of commodity computer systems. The results indicate that there are significant similarities and variations in the fault and error sensitivities between different versions of the same type of computer systems, and between different types of data in the same computer system. Furthermore, we show that recovery from errors in a large portion of static and dynamic memory space can be accomplished through simple software techniques (e.g., reloading of data from a disk). The recoverable data include pages filled with identical values (e.g., "0") and pages loaded from files unmodified during the computation. Consequently, the selection of targets for protection should be based on knowledge of recoverability rather than on error sensitivity alone.[12]

4.1. Motivation

Fault injection (FI) is a well-established experimental methodology for validating, characterizing, and evaluating fault-tolerant computer systems. The most effective reliability characterization metrics used in FI are: fault sensitivity (FS) and error sensitivity (ES), formulated in Eq. (4.1) and (4.2), respectively. FS shows the sensitivity of a computer system to a set of injected faults, and ES shows the sensitivity of a system only to a set of activated faults (i.e., errors).

$$Fault\ Sensitivity = \frac{Number\ of\ observed\ failures}{Number\ of\ injected\ faults} = \frac{Number\ of\ failures}{Number\ of\ faults} \quad (4.1)$$

$$Error\ Sensitivity = \frac{Number\ of\ observed\ failures}{Number\ of\ activated\ faults} = \frac{Number\ of\ failures}{Number\ of\ errors} \quad (4.2)$$

[12]Part of this chapter was published: K. S. Yim, Z. Kalbarczyk, and R. K. Iyer, "Measurement-based analysis of fault and error sensitivities of dynamic memory," in *Proceedings of the International Conference on Dependable Systems and Networks*, pp. 431-436, 2010.

FI experiments were conducted in various system abstraction layers (e.g., device-physics, circuits, chips, and systems). FI experiments conducted in multiple system abstraction layers (e.g., hierarchical FI [ZIR+99]) help us to better understand error propagation paths and probabilities in target computer systems (e.g., the impact of a low-layer error on the entire system).

In order to validate the findings of previous FI experiments on the latest commodity computer systems and to find universal patterns in error propagation, in this chapter, we seek answers to the following three questions:

- How do the FS and ES of different versions of the same type of computer systems change over time (e.g., over 3 years)?
- How much difference in the FS and ES is there between different types of data (e.g., dynamic memory and general-purpose register) stored in the same computer system?

To answer those questions, we developed a unified FI framework and used one FI experiment method to conduct FI experiments on multiple computer systems. Our *Extensible Fault Injection (EFI)* framework separates the controller and fault injectors. The controller thus can control multiple fault injectors installed in different target systems and apply the same FI experiment method to all the controlled target systems. The controller software has an extensible architecture that makes it easy to design and automate complex FI experiment campaigns.

In order to inject faults into various types of program states, EFI implements a data-type-aware FI technique. That data-type-aware technique allows us to classify FI results as a function of the type of corrupted data stored in either static or dynamic memory. For example, it was difficult to identify the type of the corrupted data stored in a specific dynamic memory address at the time of fault injection and/or at the time of system failure, because a dynamic memory object (e.g., a heap object) is allocated and freed at runtime. Faults randomly injected into the entire dynamic memory space [GKI04] can demonstrate the average FS and ES of the evaluated memory space but cannot characterize the ES of specific memory objects allocated in the dynamic memory.

In order to be more precise, one must be able to: keep track of dynamic objects that are allocated and freed over time, and associate a specific data type with a memory word inside a dynamically allocated memory object. The presented data-type-aware FI technique uses a symbolic identifier to specify an FI target (e.g., specifying the data type of an FI target), converts a symbolic identifier to a virtual address at runtime, and uses the converted virtual address to set a breakpoint when a breakpoint-based injector is used.

Based on EFI and the data-type-aware FI technique, we conducted the following three FI experiments.

First, we conducted FI experiments on three different versions of the Linux OS and x86 platforms. We found that the FS and ES changed significantly even when faults were injected into the same type of data on the same type of system. For example, as the OS kernel evolved, the FS of code memory was reduced because of the increase in its code size, and the ES of code memory rose because of the addition of built-in error detectors. That shows the importance of software (e.g., runtime error-checking code and compiler code generation) for system reliability.

Second, we conducted FI and profiling experiments on various types of dynamic memory objects of a Linux system. Our FI experiments showed that the variations in the FS and ES between different types of data are significant even in the same system. Specifically, dynamic memory exhibits about 18 times greater fault sensitivity than static memory does, mainly because of the higher activation rate. That high fault sensitivity implies that selective memory protection techniques that use fault sensitivity to select protection targets can incur a large overhead if applied to dynamic memory space.

Recoverability-driven memory protection can reduce protection costs under a high fault rate. Our study indicated that it is possible to recover from a significant percentage of memory errors (70% of the static memory and about 10-60% of dynamic memory) through simple software techniques. For example, a large proportion of memory pages allocated by applications are filled by the same value (e.g., "0"). Errors in these pages can be recovered from if the value is recorded in advance. Memory pages used for disk caches are replicated by default to stable storage. Furthermore, some of the user-level state can be excluded as protection targets. For example, errors in multimedia data are short-lived and quickly removed by new-incoming data (e.g., the next video frame) without degradation of the quality of service. Our evaluation data shows that this recoverability-driven memory protection can handle multi-bit errors in a cost-effective way.

The rest of this chapter is organized as follows. Section 4.2 reviews related works. Section 4.3 presents the extensible FI framework for multiple computer systems. Section 4.4 describes the data-type-aware FI technique. Section 4.5 summarizes an experiment that used three versions of the same type of computer system, and another experiment in which faults were injected in various types of program states (e.g., dynamic memory). Section 4.6 presents a recoverability-driven protection technique for memory. Section 4.7 describes a case study that shows the extensibility of the presented framework, and Section 4.8 summarizes.

4.2. Related Work

This section reviews the related works.

4.2.1. Fault Injection Tools and Experiments

Although extensive FI experimentation has been done, not much work has used the same FI method to conduct FI experiments on multiple computer systems.

(i) Data-type-aware FI. Previous fault injection studies have accurately tracked the impacts of low-layer hardware faults on the reliability and data integrity of upper-layer and larger-scale systems.

All the previous experiments on different system layers showed that not all faults in a layer can propagate and corrupt states that are visible to upper layers. The actual masked fault ratio varies depending on various configuration parameters. Historically, FI experiments have been done in a bottom-up manner, as described next.

(a) Device-physics. The early fault injection experiments in the device-physics layer used simulated fault injectors in order to model faults accurately (e.g., a double exponential model [Mes82] and an iterative simulation model [JIH+97]). Based on such models, the impact of low-layer faults on higher layers of system abstractions was studied.

(b) Circuit. In the circuit layer, in order to emulate the output of a faulty circuit in a logic layer model, the mixed-mode simulation approach [CI92][IT96] uses a fault model of a faulty part of the circuit; the model is derived in the device-physics layer. In an alternative time-dimensional approach, the model of a faulty circuit is dynamically switched during a time interval that surrounds the fault occurrence [Yan92]. In those two approaches, the cost of supporting multi-level simulations is non-negligible in terms of the simulation computing power. That technical issue is addressed in the fault dictionary approach, which uses low-layer fault dictionaries to emulate low-layer fault induced errors in a higher layer [CI92][BGG+02][KKA95].

(c) Chip. The chip layer fault injection experiments used a coarse-grained simulation model (e.g., the hardware module was treated as normal or faulty [GIY97]), the HDL code of a chip to embed fault injection code [KIR+99][LRS+08], or an off-chip pin-level fault injector [GJ95]. *(d) System*. The system layer fault injection experiments used either cycle-accurate or approximate system simulators, or software-implemented fault injectors.

In the early low-layer FI experiments, it was natural to classify the fault injection results as a function of hardware fault characteristics (e.g., angle and energy of particle strike, or location or type of a faulty circuit). That could reveal fault-sensitive (e.g., de-

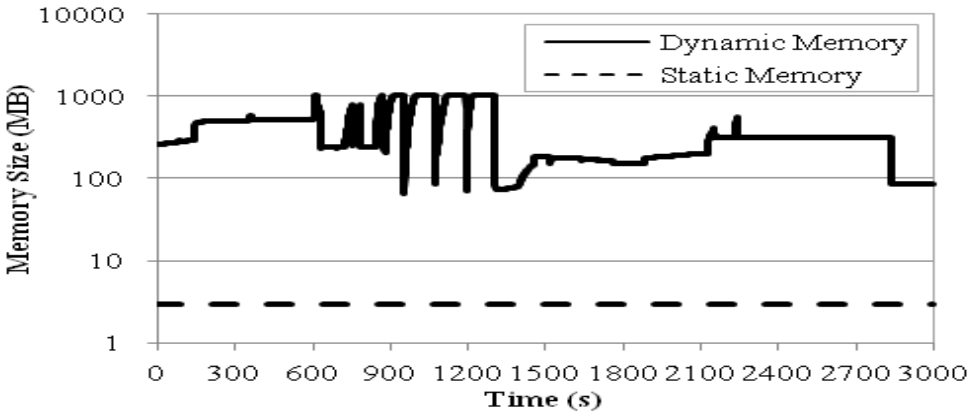


Figure 4.1. Static vs. Dynamic memory size.

coder and reorder buffer) and less fault-sensitive (e.g., instruction register and register allocation table) hardware components by, for example, analyzing the architectural vulnerability factor (AVF) [MWE+03]. The analyzed ratio is the average sensitivity of the benchmark software to injected faults in specific hardware locations. Some hardware components store and/or manipulate various types of data (e.g., register file and ALU). One of our experiments was designed to collect data necessary to analyze the sensitivities of injected faults as a function not only of the fault location but also of the type of corrupted data.

(ii) Dynamic memory FI. Many fault injection experiments have been conducted to measure and identify fault-sensitive memory states. While the error sensitivity of static memory has been well studied (e.g., using virtual addresses derived from debugging symbols of a target program binary) [GKI04][YKI09], the error sensitivity of dynamic memory has not been adequately characterized. Yet dynamic memory is, on average, about two orders of magnitude larger than static memory, according to our measurement of a regular Linux-based system; see Figure 4.1. That measurement ran the LTP (Linux Test Project) v2.0 benchmark program on an x86 machine.

If dynamic memory has been studied, faults were randomly injected into all or specific parts of a dynamic memory space. Thus, there has been almost no analysis of the correlation between data type and error sensitivity in dynamic memory space. An exception is the work described in [CKV+04], which analyzes the error sensitivity of various types of heap objects used in a JVM. This work shows the tradeoff between protection granularity and overhead. We present a similar technique that can analyze the error sensitivities of specific types of data in dynamic memory space, but our technique also supports dynamic

memory spaces of the OS kernel and user programs written in any programming language (e.g., C/C++). Of all the previous approaches, Valgrind[13] is the one most similar to our presented data-type-aware fault injection tool for OS and user-level dynamic memory objects. The main differences between those two are the granularities of object monitoring, the types of objects monitored, and the design complexity of the framework.

4.2.2. Profiling Tools

There are many existing profiling tools that are useful to perform a post-mortem analysis (e.g., anomaly-based detection). We classify the existing profilers into three types:

(i) Breakpoint-based technique. This type of techniques uses a breakpoint hardware common in modern microprocessors. A breakpoint is set at a piece of a code that is executed when a specific type of event of interest is executing (e.g., system call entry function). When a target instruction is being executed, a breakpoint exception is raised that is intercepted to execute user provided codes (e.g., to count the event occurrences). A number of hardware-implemented breakpoints are supported for each kernel and user-space breakpoint target. This type of technique is pluggable in the sense that it neither needs source code modification nor recompilation. However, it can incur a large overhead if the profiling target is executed frequently, mainly due to breakpoint exception handling overhead.

(ii) Dynamic binary rewriting. This type of techniques provides similar profiling power to the embedded hook techniques, but it does not need recompilation, as such changes are made dynamically. Specifically, a technique can look all callers of a profiling target function by using the symbol information of a target OS kernel image. This technique then rewrites the found call instructions to make them call another profiling wrapper function. This wrapper function performs specific profiling operations before calling the original target function. In modern OS kernels and processors, if the code segments are protected (e.g., read-only), the page table needs to be modified before this dynamic binary rewriting operation. After the profiling, the original instructions are restored, and thus no more performance overhead is incurred. The performance overhead of this approach is calling an extra function and running the profiling operations, while that of the embedded hook is checking a list of function pointers, calling the registered functions, and running the profiling operations.

[13] Valgrind is an instrumentation framework for building dynamic analysis tools. This project is hosted at http://valgrind.org.

(iii) Performance counter. This type of technique uses the performance counter hardware in modern processors to monitor a specific type of architectural events (e.g., cache misses and breach prediction misses) and counts them.

4.3. Tool

We present the Extensible Fault Injection (EFI) framework to support FI experiments on multiple computer systems.

4.3.1. Architecture

Figure 4.2 shows the hardware and software architecture of the EFI framework. The framework consists of three types of computer nodes: control, monitor, and injector. The *control node* is an interface between a user and injector nodes. In each *injector node*, the actual fault injection is done; that includes collection and reporting of injection results to the control node. Each *monitor node* is used to relay fault injection commands and results between the control node and a set of injector nodes. Relayed data can be preprocessed by the monitored node to prevent the centralized control node from being a performance bottleneck. The reduction in the loads of the control node enabled this offloaded processing improves the scalability of the EFI framework (e.g., simultaneous control of multiple and different types of injector nodes).

The latest implementation of the EFI framework supports three types of fault injectors: breakpoint-based injectors for CPU-based platforms (e.g., Linux OS on x86 processor, Solaris on SPARC, and AIX on POWER), and a source code mutation-based injector for GPU-based platforms (e.g., CUDA[14] programs on NVIDIA GPUs). Support of such diverse target systems is feasible thanks to the following architectural properties of EFI.

(i) Separation of control and injection. In the EFI framework, fault injection and its control operations are split and located in different types of computer nodes. The control node controls injector nodes by using application-layer protocols that run on top of the standard TCP/IP protocol (e.g., secure shell and secure FTP). For example, the control

[14] CUDA (Compute Unified Device Architecture) is a software platform that enables users to easily control NVIDIA's GPU devices. CUDA supports a programming language similar to C++ and types of API where the runtime API is easier to use than the driver API, which provides more control over GPU devices. CUDA compiler translates the C++-style code into the PTX (Parallel Thread eXecution) pseudo-assembly language code, which is eventually translated into a binary to run on GPUs. Currently, CUDA supports only NVIDIA GPUs unlike to OpenCL (Open Computing Language) that supports AMD and NVIDIA GPUs, and x86-family and POWER7 CPUs.

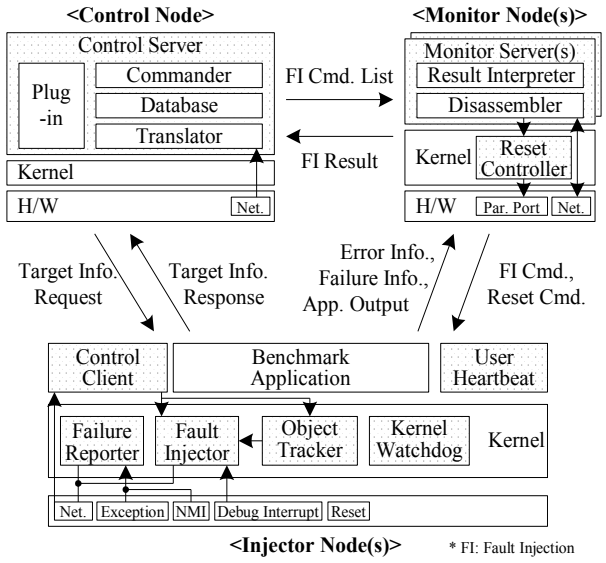


Figure 4.2. Control architecture of the *Extensible Fault Injection* framework.

node sends shell commands to injector nodes, receives the command execution result messages, and exchanges regular files with injector nodes.

(ii) Customizable controller software. The *control server* is the core control software that runs on the control node. The control server has four types of software components (see Figure 4.2). *Commander* generates a fault injection campaign and performs a fault injection experiment by processing the campaign. The collected fault injection result data are analyzed and stored in the *database* component. The *translator* component is used to translate the source code of the target software in order to embed error injection code in the source codes. It is used only for mutation-based injection in GPU-based injector nodes for which breakpoint-based injection is not feasible.

All three types of components in the control server can be customizable by the other type of components (plug-ins). A *plug-in* is a user-provided executable script. A plug-in includes the definitions of a set of commands for the *commander*. These defined commands can specify a fault injection target, start a benchmark program, collect fault injection result data, and clean up an injector node, for example. Each of those commands is a mixed sequence of shell commands, file transfer operations, and the control server plug-in programming interface (PPI) operations. PPI operations are commonly used mechanisms for conducting fault injection experiments and are served by the control server

software framework[15]. An example control scenario could be one in which a file transfer operation copies a fault injection result file from an injector node to the local file system of a monitor node, a shell command controls a monitor server to interpret the saved result file, and a PPI operation parses and saves the interpreted data in the database component on the control server.

4.3.2. Breakpoint-Based Fault Injector

Both the OS and application software maintain large runtime states for various types of data. Different types of data are likely to show different failure behaviors when the data are corrupted by hardware faults. It is thus necessary to classify fault injection results as a function of corrupted data type, except for types of data that are negligibly small from a statistical point of view. All EFI fault injectors are designed to support *data-type-aware* fault injection; the exact set of supported data types depends on the type of fault injector.

The EFI breakpoint-based fault injector can inject faults into OS and user programs on commodity CPUs. It does so with the *breakpoint-based fault injector module*, which resides in the OS kernel. When the OS kernel is hosted by a hypervisor, the same fault injector module can be used, although the performance overhead of fault injection operation varies depending on how the used hardware- or software- breakpoint mechanism is virtualized. That implies that EFI would have different performance overheads on type-I and type-II hypervisors (e.g., hardware-assisted virtualization vs. binary translation).

Breakpoint-based fault injection uses a hardware- or software-breakpoint mechanism in order to obtain the control of a target system when a fault injection target is being accessed. Users can configure the fault injector module by using shell commands or file I/O operations (e.g., *proc* file system interface[16] in Linux). The configuration command includes the target process identifier, breakpoint address, injection target type, injection target address, and error bitmask. The breakpoint-based *fault injector* module in an injector node (see Figure 4.2) sets a breakpoint on a specific virtual address. If the breakpoint

[15] We use the Jython programming language for plugins in our current implementation. Jython has a simple syntax (similar to the wide used Python script) and its programs are dynamically linkable to the EFI controller software written in Java. Thus, the plugin programming interface call is implemented as a procedure call from the Jython plugin script to the externally exposed Java methods of the controller software. The Jython and Python projects are at http://jython.org and http://python.org, respectively.

[16] Proc file system (*procfs*) is a special-purpose file system in UNIX-variant OSes that dynamically provides information about resources managed by the OS (e.g., hardware, kernel, and user process resources). Each *procfs* is organized as a conventional hierarchical file system structure, and is typically mapped under the */proc* directory of the root file system.

is triggered, the breakpoint handler emulates a soft error in a target system state. Injections into the following data types are supported for both kernel- and user-level software.

(i) Processor register. The breakpoint-based fault injector can inject faults into general-purpose control and data registers as well as special-purpose registers in processors. A breakpoint is set on an instruction in the code memory of the virtual address space of a target process (or any process if the target is the OS kernel for example because the Linux kernel resides in the 4th gigabyte of virtual address space and the kernel space is shared by all processes in the system). We use either a hardware breakpoint feature of a processor or a software-breakpoint mechanism. Software breakpoint is implemented through dynamic rewriting of the OS kernel code for kernel-level injection, or through use of *ptrace*[17] system calls for user-level injection. The fault injector module uses the context of a target OS kernel or user process. The context is saved at the entry of every interrupt handling event. Such context information (e.g., general-purpose register values) is stored in the kernel stack or the process control block (PCB) of a preempted process, depending on the OS implementation. The breakpoint handler of the fault injector module emulates an error by modifying the target register value saved in the stack or the PCB. The corrupted context is restored to the processor hardware register just before returning from the breakpoint handler to the preempted target process. Thus, when the process resumes, the corrupted register value is visible to and can be used by the preempted target process. If the fault injection target is one of certain special-purpose registers (e.g., an MMX register in x86), the target register value is directly modified by the breakpoint handler, because many special-purpose registers are neither saved at the entry of an interrupt handler nor modified inside an interrupt handler.

(ii) Memory data. In memory, the supported injection targets are the code, static data, dynamic data, and stack memory segments. A hardware breakpoint is set on the virtual address of a target data and is triggered before the target address is accessed for read or write. The breakpoint handler changes the target memory value (e.g., by using a provided error bitmask) in order to emulate the effect of a fault. If the target is part of statically allocated memory spaces, the symbol information of the binary of a target program or the image of an OS kernel is used to identify the actual type of data or instruction stored in the target virtual address.

[17] *ptrace* (Process Trace) is a system call that allows the caller process not only to control and monitor the execution of another process but also to inspect and manipulate the internal state of another process. This is common in most UNIX-variant OSes that includes Linux because this provides the key mechanism for software debuggers.

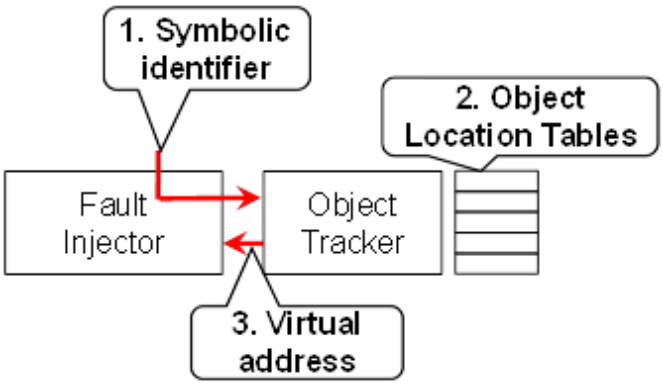


Figure 4.3. Selection process of fault injection target using object tracker.

4.4. Measurement Method

When the data-type-aware FI is being realized, the main technical difficulty comes from FI targets that are in dynamically allocated resources. We present a data-type-aware fault injection technique for dynamic memory data. This technique consists of: (i) the *object tracker* module (see Figure 4.2) that tracks dynamic memory objects and translates a symbolic identifier (that specifies the type of a memory object) to a virtual address, and (ii) the *profiler module* that monitors the dynamic memory regions (e.g., size and read/write ratio).

4.4.1. Object Tracker

The *object tracker* in each injector node is used to select a fault injection target in dynamic memory space. The object tracker tracks various types of dynamic memory objects. The tracked information stored in the object location table and the OS kernel memory is used to translate the *symbolic identifier* of a fault injection target to the corresponding virtual address (see Figure 4.2). Symbolic identifier specifies the data type of an injection target and the index of an instance of data of the targeted type. Note that when the object tracker is used, the control server sends a symbolic identifier instead of a virtual address to specify a fault injection target.

The object tracker has three different tracking granularities:

(i) Page granularity. The object tracker tracks all physical pages belonging to each memory region. The following extension is made to calculate the virtual address of a specific page in a memory region. We extend the metadata of a physical page frame with

Table 4.1. Instrumentations to track allocation/free in memory region.

Region	Event	Instrumentation Location
Buffer Cache	Grow	drivers/md/raid5.c: grow_buffers(…)
	Shrink	drivers/md/raid5.c: shrink_buffers(…)
Page Cache	Grow	mm/filemap.c: add_to_page_cache(…) mm/page_alloc.c: page_alloc_cpu_notify(…)
	Shrink	mm/filemap.c: __remove_from_page_cache_nocheck(…)
Swap Cache	Grow	mm/swap_state.c: __add_to_swap_cache(…)
	Shrink	mm/swap_state.c: __delete_from_swap_cache_nocheck(…)
Anon	Grow	mm/rmap.c: __page_set_anon_rmap(…) mm/rmap.c: page_add_file_rmap(…)
	Shrink	mm/rmap.c: page_remove_rmap(…) if PageAnon(page)
Mapped	Grow	mm/highmem.c: kmap_high(…)
	Shrink	mm/rmap.c: page_remove_rmap(…) if !PageAnon(page)
Page Table	Grow	arch/i386/mm/pgtable.c: pte_alloc_one_kernel(…) arch/i386/mm/pgtable.c: pte_alloc_one(…)
	Shrink	include/asm-i386/pgalloc.h: pte_free_kernel(…) include/asm-i386/pgalloc.h: pte_free(…)

two fields: *type of memory region* and a *node for a linked list* (i.e., maintained for all page frames in each memory region). In page-granularity tracking, the symbolic identifier consists of the region name and the index of a page in the region (i.e., allocation order). Through searching of the linked list of a target region, the page frame structure is obtained, and the virtual address of the page is computed. For regions directly belonging to a specific memory allocator, the list is maintained by the allocation and free functions of the memory allocator. For the rest of the pages whose region type changes over time, we instrument the functions that can change the region type. Table 4.1 summarizes the instrumentation points.[18]

[18]UNIX-variant OSes use similar dynamic memory management techniques. Linux kernel has four types of dynamic memory allocators (see Figure 4.4) [BC00][Vah96]: *(i) Buddy allocator*. The buddy allocator is the top-level allocator for physical memory. It (de)allocates a contiguous physical memory space that has a multiple of a page frame (e.g., 4 kB). The other allocators rely on this allocator to obtain page frames and return the obtained frames. The buddy allocator is used for cache regions. The page cache region contains pages originating from files. The buffer cache region keeps pages that are being transmitted from/to a storage device. The function of the swap cache region is to hold pages read from swap areas where swap area is to keep pages evicted from the memory due to a memory overflow. This allocator is also directly used for the page table region when allocating pages for page table entries. *(ii) Slab allocator*. The slab allocator reduces the overhead of the buddy system when small-size objects are frequently allocated and freed. It has a set of slab caches where a cache keeps a set of slab objects that have the same size. For example, a slab cache can store up to 128 32 bytes objects by using a 4 kB page frame. This solves the internal fragmentation problem of the buddy allocator. This allocator serves the kmalloc allocator and the slab region that contains the common kernel data structures. A part of the page table region also uses the slab allocator (e.g., page global/middle directory). *(iii) Kmalloc*. The kmalloc allocator is useful in managing variable-size

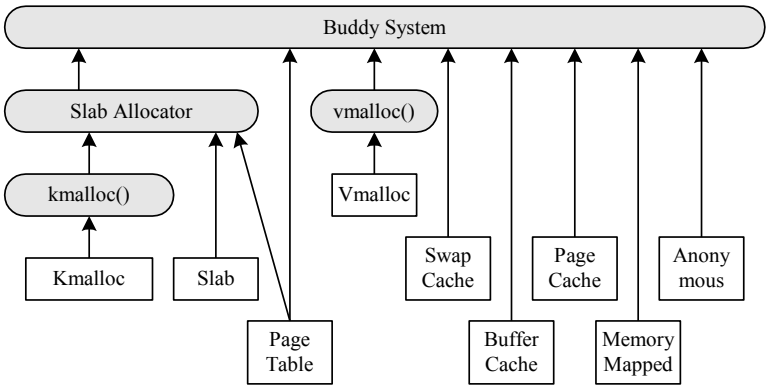


Figure 4.4. Dynamic memory allocators and regions in Linux OS.

(ii) Object granularity. The object tracker classifies the tracked dynamic memory objects by using the type of memory allocator and the call stack signature of a function that called the allocator function. For example, a memory object allocated for kernel modules can be specified by *sys_init_module()* as the first-level caller of the *vmalloc()* allocator. In our framework, up to 30 nested callers can be specified to point to an object type. In >95 % of cases, the nest call depth from the system call entry is smaller than 30 according to our measurement experiment. This call stack signature is obtained through tracking of the function frame pointers. Specifically, in x86 ISA, the program counter (EIP) and frame pointer (EBP) registers are stored in the call stack when call instructions are executed. EBP points to the old EBP in the stack. Because the old EIP of the caller is stored in the 4 bytes above the old EBP, the virtual address of the caller function is obtained. Through searching of the symbol tables of the kernel and modules with the old EIP, the symbol name of the caller is obtained. That search is repeated up to 30 times until the bottom of the stack is reached.

All allocation and free functions are instrumented. The instrumentation routines (which are a set of pairs of allocator and free functions) are enabled to extract the caller

---

memory objects. Its interface is similar to malloc() and free(). Internally, it uses the slab allocator and creates a set of slab caches where the object sizes of the caches are geometrically distributed from 32 bytes to 128 kB. If an object is requested, it forwards the request to a slab cache that best fits into the requested object size. *(iv) Vmalloc*. The vmalloc allocator can allocate variable-size memory objects that are contiguously allocated in the virtual address space but not always in the physical address space. It gets page frames directly from the buddy allocator and maps the page frames into a contiguous virtual address region. It serves as the vmalloc region, containing buffers to copy code pages from files to kernel modules and I/O buffers for some device drivers and file systems. In addition, two more memory regions are identified in the dynamic memory of Linux. The memory-mapped region contains pages mapped into the last 128 MB of virtual address space, into which the physical memory above 896 MB is dynamically mapped. The anonymous region contains pages for user-level data.

signature and to find an object that belongs to the specified object type. When an instrumentation routine finds the specified object, the allocated virtual address is sent to the injector for fault injection. If the object is freed before the breakpoint (set by the injector) fires, the breakpoint is unset, and the control client (executing in the user space) is notified of the event.

(iii) Variable granularity. Variable granularity tracking is realized by analyzing the source code of callers. The analysis extracts the data types of internal variables of a target memory object. The extracted information is used to match the internal variable type to the offset in a tracked memory object. For memory regions using the slab allocator, variable-granularity tracking is easily implemented. The symbolic identifier is the name of a slab cache and an object index. All active slab objects in a specified slab cache are identified through scanning of the data structure of the slab cache in the OS kernel memory, and the index is used to select a specific object. The offset inside the object reveals the variable type because the object data type for a slab cache is fixed. The source code analysis (considering the memory alignment by the compiler) is used to match the offset with the variable type.

4.4.2. Profiler

We developed three memory profiling techniques to support the data-type-aware fault injection and the accurate measurements of error sensitivity and the sizes of memory regions:

(i) Memory allocation/free behavior. The profiler stores the number of all memory allocation and free events for all monitored caller signatures. A static memory area is used to store these event counts. Each entry in the static memory stores the caller signature, a parameter of the innermost call, and the total call count. In addition, the profiler keeps a set of variables to track the sizes of allocated memory regions. Upon a request from the user (e.g., upon an application completion), the valid entries are stored in a log file. A kernel thread is used to periodically (e.g., every second) store the profiled information to a log file while a benchmark program is being executed. That logging allows us to capture the variances of profiled information over time.

(ii) Activation and read/write ratios. To measure the activation and read/write ratios of a target state, we capture all read and write operations performed on the target state. We use a breakpoint that is fired by either a read or write operation. When the breakpoint fires, the instruction byte code (up to 8 bytes in x86 ISA) pointed to by the current program counter is sent to the monitor server (or to the control server if the monitor server is

unused). The monitor server disassembles the instruction byte code and analyzes the operands to determine the type of memory operation (e.g., read or write).

(iii) Searching pages from the entire physical memory. We have implemented a kernel feature that scans the entire physical memory and finds all memory pages for which the entire page is filled by the same value. Both the kernel image memory (that loads the image of a Linux ELF file) and the address spaces maintained by the *kclist* linked list are scanned in Linux. The scanned areas are the same as the areas scanned by the kernel core dump module (*/proc/kcore* file in Linux). Our method reduces the interference on the memory usage pattern because our method does not copy the scanned memory to another memory location or a file.

4.5. Experiment: Sensitivity

We selected the Linux system and evaluated the FS and ES of three different versions of that system running on the same processor (i.e., Pentium 4). The used kernel versions were v2.4.22 (released at August 2003), v2.6.5 (April 2004), and v2.6.16 (May 2006). All faults were injected into the kernel code memory.

Observation 4.1. *In all three selected version of a specific type of commodity computer system, large percentages of the transient hardware faults that propagated to the architectural states (e.g., ~62.6–72.8% for code memory) were benign from the point of view of software reliability (see Table 4.2).*

Our analysis concluded that the fault masking effect was mainly due to the following causes:

- *Unactivated faults.* A large portion of memory data is not used by programs. For example, the execution of an instruction in code memory is determined by the program control flows. Some program functions and basic blocks are rarely or never called under specific program input data. That is especially true in the OS kernel because its control flows depend heavily on the occurrence order and time of asynchronous events, and the I/O data content (e.g., file system metadata). Similarly, strong temporal and spatial localities in the access patterns of data memory disprove the existence of less-frequently accessed memory data.
- *Overwritten.* If the first access to the corrupted data in the register or memory is a write operation, this error is removed from the system before it can propagate to any other state. That is common in data memory. It is still possible in code

Table 4.2. Fault sensitivity vs. Linux kernel version.

Fault Location	Not Activated	Not Manifested	User Failure	OS Failure	Target System OS	ES
Code Memory	45%	17.6%	0.6%	36.9%	Linux 2.4	68.1%
	54%	15.2%	0%	30.8%	Linux 2.6	67.0%
	64.7%	8.1%	0.1%	27.1%	Linux 2.6	77.1%

memory, for example, it can result from the use of dynamic binary rewriting and self-modifying code.

- *No impact on program execution behavior.* If none of the uses of some corrupted data change the program behavior, this error is not manifested. For example, the result of the ($x > y$) condition does not change if x is corrupted from a value larger than y to another value that is still larger than y.
- *No data flow to externally-visible data.* When a program execution behavior is changed and an incorrect state is created, it causes a user-visible failure only when the corrupted state propagates to program output data or shows up as an externally visible symptom (e.g., a crash due to an access to an invalid memory address).

We then analyze the ES of the same three systems.

Observation 4.2. *All three systems have relatively similar error sensitivities when the same type of transient hardware faults is injected. 58–67% of the activated faults caused failures in the code memory.*

As the kernel evolved, the FS lowered, mainly because of the lower fault activation ratio (see Table 4.2). The reduction in the fault activation ratio was mainly a result of the increase in kernel code size (e.g., diverse kernel features and a rich set of device drivers). On the other hand, as the kernel evolved, the ES rose, mainly because of the addition of built-in error detection codes (e.g., assertion) in the kernel, which meant that a much larger portion of injected faults were caught.

Observation 4.3. *As the OS kernel evolves, the fault sensitivity to code memory faults goes down because of the increase in kernel code size; the error sensitivity to code memory faults goes up because of the continuous addition of built-in error detectors; and the hang failure ratio goes up because of the use of symmetric multiprocessing (SMP) kernels and many more processor cores.*

Table 4.3. Control data vs. Non-control data.

Fault Location	Data Type	Not Activated	Not Manifested	User Failure	OS Failure	Fault Sensitivity	Error Sensitivity
Memory	Code	64.7%	8.1%	0.1%	27.1%	27.2%	77.1%
	Data	95.3%	1.6%	0.4%	2.7%	3.1%	66.0%
Register	Control	N/A	22.8%	6.1%	71.1%	77.2%	77%
	Data	N/A	79.9%	2.0%	18.1%	20.1%	20.1%

4.5.2. Control vs. Non-Control Data

We characterize faults in a Linux-based system on an x86 machine as a function of the corrupted data type. Evaluation of ES in various types of data naturally shows the strength of baseline error detectors in the hardware and software of modern computer systems. A large portion of errors in OS kernel programs on CPU-based systems were detected by basic error detectors.

A non-benign error is likely to be detected if an induced error propagates to control data. Control data include: (a) data that represent a call-flow state of a program (e.g., a program counter, return address, frame pointer, or stack pointer), (b) data used to make a control-flow decision (e.g., a branch condition), and (c) data used to express a memory access address (e.g., an address operand of a memory load, store, or indirect jump instruction). Because the integrity of those control data are likely to be checked by the baseline error detectors, if an error propagates to control data, it is likely to be detected before it causes a system failure. For example, if the address operand of a memory store instruction is modified to an address of an invalid page, that is detected (e.g., as a segmentation fault exception) in CPUs. If the condition of a conditional branch is flipped, that is also likely to be detected because it greatly changes the program execution behaviors (e.g., executing instructions and data that are irrelevant to each other).

There is a strong correlation between the FS and the fault activation ratio. For example, control registers have a higher sensitivity than data registers do. The reason is that the control registers (e.g., the program counter and stack pointer) are likely to contain live data, while data registers contain both live and dead variable values. Similarly, the large difference in the fault sensitivities of code and static data memories comes from the difference in their fault activation ratios (see Table 4.3). Factors that lead to a low activation ratio in data memory include weak locality (e.g., static data) and large memory size (e.g., dynamic data memory; see Figure 4.6).

Observation 4.4. *A large portion of failures are detectable by the baseline error detectors in the OS and CPU. In OS kernels, 87.9–99.1% and 90–100% of failures caused by memory and processor faults, respectively, are detected.*

Our analysis shows that the basic error detectors are strong and efficient in detecting many non-benign errors in OS kernel states. The reason is that those baseline error detectors have been carefully designed and optimized over the last several decades. Most modern processors have a set of access control rules that raise an exception if these rules are violated. For example, the MMU and TLB together monitor the access permissions for all memory accesses (i.e., read, write, and execution). User-mode programs are prohibited from executing privileged instructions. Moreover, a large portion of OS kernel code is devoted to detecting and tolerating hardware-induced errors. Assertion-based error checking is common; it checks not only the input parameters but also the software and hardware states when a system call or a function call event occurs.

4.5.3. Static vs. Dynamic Memory

Fault injection is used to characterize fault/error sensitivity of dynamic memory. Over 52,000 faults (single-bit errors) were injected into a dynamic memory region (slab region) on a Linux-based system. The target was monitored for 1 minute after each fault injection. For kernel dynamic memory, faults were injected into the slab objects with the most frequently used data types. The exact count of examined data types was chosen so as to cover at least 80% of the memory space of a region. A small number of data types or caller signatures (e.g., <10) typically form >80% of the used memory spaces (see Table 4.4). The reason is that the kernel has a fixed number of data types that are specified by humans (with limited memory). We select the 8 most frequently used data types, which cover >80% of the slab region space.

(i) Fault sensitivity. Figure 4.5 shows the fault sensitivity of the most frequently used dynamic memory objects. The y-axis in Figure 4.5 is truncated at 30% because the remaining 70% corresponds to unactivated faults. The failure type overwritten means activated faults for which the first access was a memory-write; not manifested refers to an activated benign fault for which the first access was a memory-read; and disk corruption refers to file system corruption.

Observation 4.5. *The fault sensitivity of dynamic memory is much higher than that of static memory (e.g., 5.83% vs. 0.32%) mainly because of the higher fault activation ratio of dynamic memory (e.g., ~16.7 times higher).*

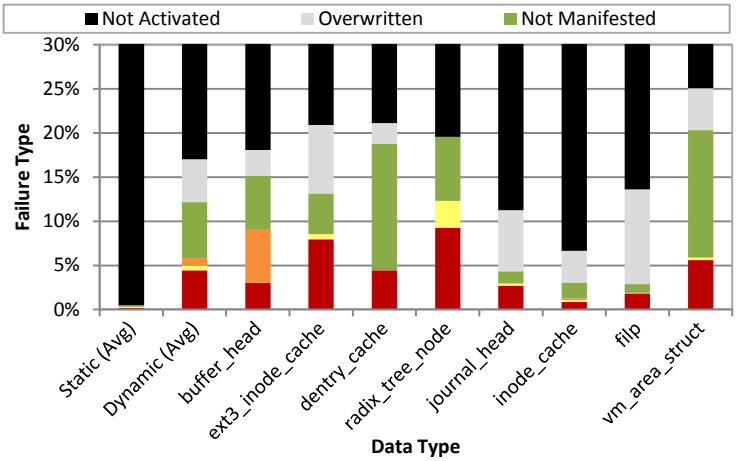


Figure 4.5. Fault sensitivity of static vs. dynamic memory space.

For comparison purposes, we compute the fault sensitivity of the static memory based on our earlier work on error characterization of a Linux kernel [GKI04]. In that study, we measured a fault activation ratio of 0.5% when we inserted faults into the kernel data segment. *For faults in dynamic memory, analyzed in this chapter, we observed a much higher fault activation ratio of 16.7% on average*. The reason is that dynamic memory pages are managed by a locality-aware (e.g., LRU-variant) cache replacement algorithm, while static memory is placed in a fixed memory location regardless of its access frequency. *That high fault activation ratio leads to fault sensitivity of dynamic memory that is 18 times higher than that of static memory (5.83% vs. 0.32%).*

(ii) Error sensitivity. In dynamic memory, 34.8% of activated faults manifest, while in static memory, 65.9% of activated faults manifest (based on [GKI04]). Furthermore, in dynamic memory, 44.7% of not manifested faults have a memory-write operation as their first access. After excluding those overwritten faults, we find that 49.2% (= 5.8%/11.8%) of faults activated by read operations caused failures. That is lower than the ratio measured in static memory (65.9%), where the read/write ratio of not manifested faults in static memory has not been reported [GKI04].

Observation 4.6. *The error sensitivities of static and dynamic memories are more similar (e.g., 49.2% vs. <65.9%) than their fault sensitivities are.*

The relatively large variation in the error sensitivities of different dynamic memory objects makes it difficult to directly compare the average error sensitivities of dynamic memory and static memory. Specifically, we injected 800 faults into the *inode* structure one by one. There was a clear difference in the error sensitivities depending on the data

Table 4.4. Most frequently used slab caches.

Slab Cache Name	Data Type		Slab Cache Size		
	Data Type Name	Data Size	Avg.	Stddev.	Percent
buffer_head	struct buffer_head	52B	3.39 MB	3.20	33.7%
ext3_inode_cache	struct ext3_inode_info	544B	1.50 MB	0.27	14.9%
dentry_cache	struct dentry	136B	1.23 MB	0.53	12.2%
radix_tree_node	struct radix_tree_node	276B	0.97 MB	0.39	9.7%
journal_head	struct journal_head	52B	0.34 MB	0.94	3.4%
inode_cache	struct inode	380B	0.32 MB	0.06	3.1%
filp	struct file	192B	0.20 MB	0.02	1.9%
vm_area_struct	struct vm_area_struct	88B	0.18 MB	0.11	1.9%
SUM	-	-	8.35 MB	-	80.8%

types of the variables. For example, pointer variables for linked lists or associated objects (e.g., the *next*, *a_ops*, and *backing_dev_info* fields) had error sensitivities higher than 50%. Furthermore, lock variables (e.g., the *break_lock* field of the *rwlock_t* data type in the *inode* structure) had similarly high error sensitivities. The fact that error sensitivities of such control data are not close to 100% indicates that not only the data type but also how the data are used are important factors in determining the error sensitivities. The reason is that not all corrupted (and activated) pointers cause failures, and the *break_lock* is highly error sensitive despite being an integer variable. File system metadata (e.g., *buffer_head*) can cause serious failure if corrupted. For example, a single-bit error in a *buffer_head* object can change the file system to a read-only mode and corrupt the file system, which potentially can be recovered via *fsck* at the next boot-up of the system.

4.5.4. Modeling

We model the error sensitivity (i.e., the probability that an error is benign $P_{benign}(f_{lo})$) as a function of the fault location.

We observe that measuring fault sensitivity is simpler than measuring error sensitivity. Below we derive an analytical expression (see (4.3)) that gives error sensitivity (*ES*) as a function of: (a) fault sensitivity (*FS*), (b) probability of fault activation (P_a), and (c) the ratio of read access count to read/write access count (P_r). Equation (4.3) also captures a scenario in which a fault is activated but does not manifest in the first read access, and it can be reactivated and manifest in the subsequent read accesses.

$$\begin{aligned} \mathrm{FS}(\mathrm{P_a}, \mathrm{P_r}, \mathrm{ES}) &= \mathrm{P_a P_r}\{\mathrm{ES} + (1 - \mathrm{ES})\mathrm{FS}(\mathrm{P_a}, \mathrm{P_r}, \mathrm{ES})\} \\ \Leftrightarrow \mathrm{FS} &= \frac{\mathrm{P_a P_r ES}}{(1-\mathrm{P_a P_r}+\mathrm{P_a P_r ES})} \\ \Leftrightarrow \mathrm{ES} &= \frac{\mathrm{FS}(1-\mathrm{P_a P_r})}{\mathrm{P_a P_r}(1-\mathrm{FS})} \end{aligned} \tag{4.3}$$

To explore the possibility of statically deriving P_a and P_r, we analyzed the variations of the P_a and P_r of different object types in dynamic memory over time.

(i) Variations of P_a. We found a relatively large variance in the activation ratio between different object instances of the same data type. For example, the standard deviations of the activation ratios of the *radix_tree_node* and *inode* slab objects in Linux OS were 11.6% and 7.3%, respectively. The activation ratio of *radix_tree_node* objects varied from 49% to 17% over time.

(ii) Variations of P_r. The read ratio also had a large variation between object types. The objects belonging to the *radix_tree_node* data type had a high read ratio (P_r = 93.6%) and thus a high fault sensitivity (FS = 12.39%). Using those two parameters with the activation ratio (P_a = 19.6%), we can calculate the error sensitivity of *radix_tree_node* as 62.4%. The read ratio also had a large variation over time.

Observation 4.7. *Different variables in the same type of memory object have a large variation in fault sensitivity, mainly because of large variations in the fault activation ratio and the probability of the first access is being read.*

Those two findings indicate that one must be cautious when estimating error sensitivity using statically derived parameters of P_a and P_r. For example, using simple averages may not be the best approach, and hence, experimental evaluation as discussed here remains a trustworthy alternative way to derive such parameters.

4.6. Experiment: Recoverability

We profile the dynamic memory space and present a recoverability-driven memory protection technique.

4.6.1. Software Recoverable Memory Area

Our measurements indicate that it is possible to recover from a large percentage of memory errors (70% and 10-60% of static and dynamic memory, respectively) through simple software techniques, e.g., reloading of data from permanent storage. Thus, we suggest the recoverability-driven protection principle that selectively protects system state that is not recoverable by software.

Table 4.5. Volume of static kernel segments (Unit: KB).

Segment	text	rodata	data	bss	Total
Size	1.824	0.264	0.476	0.362	2.970

(i) Static memory. Table 4.5 gives the sizes of all static kernel segments. Here, *text* refers to kernel code other than the codes of kernel modules; *rodata* stands for initialized read-only data (e.g., strings, bitmaps, and the system call table); *data* denotes read-writable data memory (e.g., symbol tables or initialized global data); and *bss* corresponds to zero-initialized data. The total size of the static kernel segments is close to 3 MB. The size depends on neither the hardware configuration nor the workloads, and it is determined by the compile-time kernel configuration (e.g., the kernel version, features, and statically linked modules).

The *text* and *rodata* segments are recoverable if the replicas are stored on a disk. Those two segments form 70.3% of static kernel segments. When an uncorrectable error is detected by MMU, the system is not shut down; instead, the location of the fault can be analyzed. If the fault is in one of those two segments, the relevant parts of the boot kernel image can be reloaded (e.g., from a disk) to remove the error. Note that computation errors cannot propagate to those segments if writes are prohibited by MMU-based memory protection.

(ii) Dynamic memory. Figure 4.6 shows the sizes of the dynamic memory regions in a log-scale. The dynamic memory size varies greatly depending on the executed workload phases. On average, the total size of the dynamic memory (e.g., 248MB) is about two orders of magnitude larger than that of the static memory. In high-end systems, the size of the dynamic memory space is close to the physical memory size because it runs multiple threads, and the size of the physical memory is optimized for the target workloads, expected performance, and memory cost. On average, the page cache, buffer cache, anonymous user, slab cache, and memory-mapped regions constitute 55.7%, 23.5%, 6.8%, 5.5%, and 5.1%, respectively, of the entire dynamic memory space. The swap cache, *vmalloc*, *kmalloc*, and page table regions occupy small memory portions (<1.2% each).

Among dynamic memory regions, we identified three sub-regions where faults can be recovered via simple software techniques. These regions keep data that are replicated on a storage device, or data initialized at allocation time but not modified during the computation.

(a) Re-computable data. It is possible to recover from an error in a page in which all words contain the same value by rewriting the known value. Figure 4.7 shows the portion of physical memory space containing memory pages filled with the same value. In our

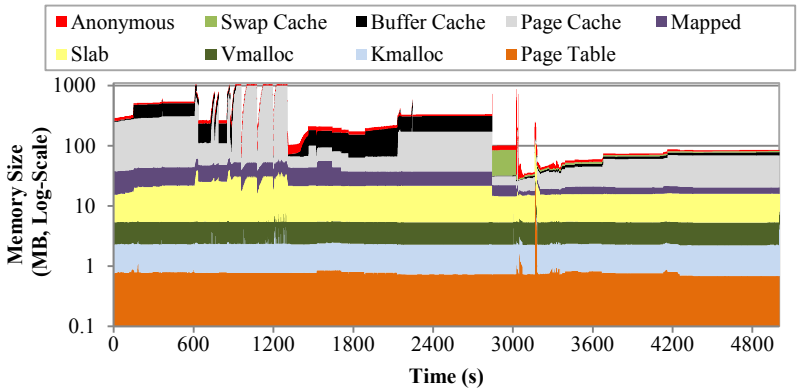


Figure 4.6. Dynamic memory space broken down by memory region type.

experiments, measurement samples were taken every three minutes during execution of LTP. The re-computable pages formed 10–60% of the physical memory space.[19] On average, 38% of memory space contained pages with a zero value written in every location over the period of 5,000s. Pages filled by other values constituted less than 1% of used memory. The dominant value patterns changed as the program executed (e.g., $30303030_{(16)}$ in Figure 4.7).

Interestingly, we found that not all zero-filled pages were unused memory; most of them were allocated and initialized by the application. Evidence of that can be found in close analysis of the data presented in Figure 4.6 and 4.7. Specifically, Figure 4.6 shows a memory overflow that occurred around 17-20 minutes into the measurement period. For the same time interval, Figure 4.7 indicates an increase in the number of zero-filled pages, meaning the application was allocating memory. As the memory was freed (see the time interval at 20–24 minutes in Figure 4.6), the number of zero-filled pages decreased (see Figure 4.7). A second piece of evidence comes from the profiler, which indicated that only 10.9MB of zero-filled memory pages were allocated using three Linux API functions that can allocate a zero-filled page, e.g., *get_zeroed_pages()*. This implies that the vast majority of zero-filled pages were initialized by the user programs that allocated the memory.

Analysis of a common program behavior explains the presence of zero-filled pages. An application program typically initializes its data structure when it is launched or starts to process a new event (e.g., a user request). The initialization includes writing of a default value to the memory, which can be done by a loader or user-level library if the ini-

[19]In some OS (e.g., AIX), identical value pages are less likely because allocation of such pages is recognized and properly handled by the OS kernel.

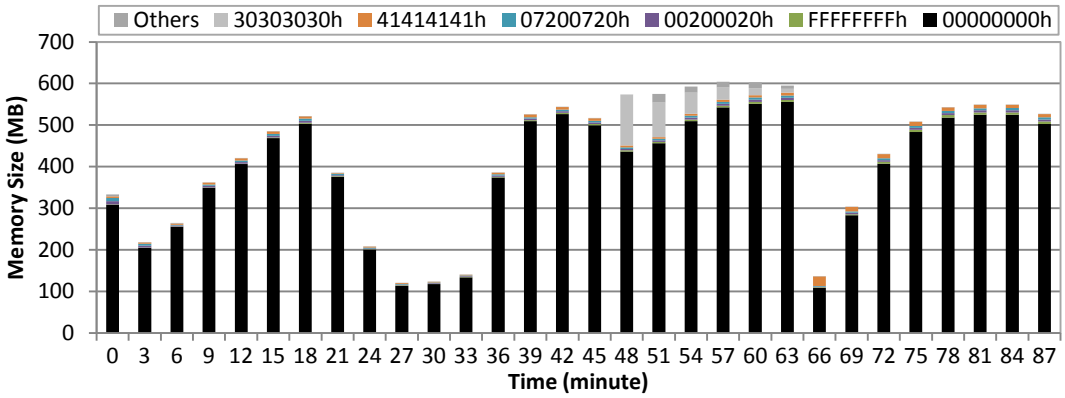


Figure 4.7. Physical pages with an identical value.

tialized data is part of a *bss* section (i.e., zero-initialized). However, the application does not use the entire space of the initialized pages. The unused pages are freed when the program finishes. Furthermore, in a page that is modified during the lifetime of a program, it can be a long time until the first modification. As a consequence, zero-filled pages can stay in memory for a long time.

In on-chip caches, a finding similar to ours (i.e., a large portion of zero-filled pages were found in memory) has been reported, and has been used as the basis for design of a cost-efficient compressed cache, as zero-filled cache lines can be compressed in a space efficient way [YZG00].

(b) Data replicated by default. It is possible to recover from errors in a memory page loaded from a storage device by reloading the storage page, if it was not modified. Like static kernel text segments, text segments for kernel modules and user programs, except for self-modifying code, also belong to this category. Those code pages were part of the page cache and *vmalloc* regions that constituted 55.7% and 1.2%, respectively, of the total dynamic memory space. Also, data pages read from a file are recoverable by software. Those data pages were mostly in the buffer cache, where the size of the buffer constituted 23.5% of the total dynamic memory in our experiment.

(iii) User multimedia data. Most single errors in user multimedia data pages are automatically removed as the computation moves forward. For users, it is difficult to recognize the degraded QoS even if 100 faults are uniformly injected into a data buffer sent to the graphics processor to compute an output image. Note that that fault rate (e.g., 100 faults per video frame) is unrealistically high. Moreover, corruptions of a single frame are not always noticeable to human eyes (e.g., for a frame rate of 100fps). Such multimedia

data are a part of an anonymous region (23.5% of the dynamic memory on the host) and constitute most of the graphics device memory space.

4.6.2. Recovery Technique

For code memory, upon an error, an extended OS can restore the kernel code from a storage device. The program code is typically read-only and is loaded from the file system at boot-up. Thus, it is sufficient if the hardware protection detects an error and triggers this basic software recovery technique.

Still many errors in both static and dynamic memory are masked (e.g., >85%) and do not harm the correctness of application software. If a hardware-implemented memory error detection technique (e.g., duplication) is used, it can trigger many unnecessary error recovery operations (e.g., checkpoint-rollback) for the benign errors, unless, for example the error sensitivity in application software is analyzed and reported to the hardware technique in a timely manner. Thus, application-guided fault tolerance techniques can optimize the performance overhead by strategically targeting non-benign errors. Application-guided fault tolerance must be carefully designed because of the relatively large variance in the fault activation ratio (a key parameter for error sensitive estimation), even between memory objects of the same data type. For example, the standard deviations of the activation ratios of the *radix_tree_node* and *inode* data structures are 11.6% and 7.3%, respectively. That means that even between memory objects of the same type, the error sensitivities can be largely different.

Our study also indicated that it is possible to recover from a significant percentage of detected memory errors (70% of static memory and about 10–60% of dynamic memory) via simple software techniques. For example, a large proportion of memory pages allocated by applications are filled with the same value (e.g., '0'). It is possible to recover from errors in such pages if the value is recorded in advance. Memory pages used for disk caches are replicated by default to stable storage. Furthermore, some of the user-level state can be excluded as protection targets. For example, errors in multimedia data are short-lived and can be quickly removed by new-incoming data (e.g., the next video frame) without degrading the quality of service. That recoverability-driven memory protection can handle multi-bit errors in a cost-effective way. Consequently, we believe that the selection of targets for protection should be based on knowledge of recoverability rather than on error sensitivity alone.

4.7. Case Study

The following case study is designed to evaluate the extensibility of the EFI controller. In this case study, the EFI controller is used to build a web-based fault injection framework. The development task has been done by a college graduate (i.e., the subject) who majored in a real science and had learned the Java programming language.

In this case study, DNA sequencing is the application domain. DNA sequencing is a process of determining the order of nucleotide bases (adenine, cytosine, guanine, and thymine) in a DNA molecule. The subject had specific interest in performing fault injection experiments on the BLAST algorithm[20] that is used as part of the DNA sequencing process and uses a nucleotide or protein sequence read by a DNA scanner[21]. However, the DNA scanner device is prone to providing corrupted scanned sequence data (e.g., insertion, deletion, and substitution errors). Although advances in DNA scanner technology have been continuously reducing the error rate, state-of-the-art devices in 2011 have error rates (e.g., >0.1% of scanned bases are incorrect) higher than the fault rates of commodity computer hardware.

One of the experimental goals of the subject is analyzing the correlations between the patterns in input sequence data errors and the accuracy of the BLAST similarity analysis results. Such analyzed correlations can help the scanner designers tune the device such that the device is not likely to cause errors sensitive to the accuracy of the BLAST analysis results and unlikely to cause errors affecting that accuracy.

Similar to other bioinformatics software, a public web service exists that provides a BLAST application and a set of integrated sequence databases. The fault injection on the BLAST input data is done by controlling a web browser. The control operations are: navigating to the target web page, entering sequence input data that contains the generated data errors, and parsing the BLAST similarity analysis results. The subject uses the PPI operations of EFI to automatically perform these control operations where the PPI operations internally use Selenium[22] (see Section 4.3 for details). The subject used the gang scheduler PPI operations of the EFI controller to concurrently run multiple fault injection

[20] BLAST (Basic Local Alignment Search Tool) finds locally similar regions between sequences. This program compares a user provided nucleotide or protein sequence to the sequences in a database of the same type and calculates the degree of statistically similarity. An web-based BLAST service is available at http://blast.ncbi.nlm.nih.gov.

[21] DNA scanner is a tool that can read the biophysical, energy, protein interaction, protein sequence, and other features from a DNA.

[22] Selenium web application testing system, http://seleniumhq.org.

experiments where the degree of concurrency is left as a user-controllable parameter. The subject also used the existing PPI operations to summarize the experiment results on database component of the EFI controller. However, the subject had to custom implement an error generator component that gets the pattern information of the data errors (e.g., error type and error rate) and a list of original sequence data and generates a list of the corrupted DNA sequences. This component was implemented as a part of the EFI controller because of the familiarity of the subject with the Java programming language. In practice, such an extension can be made by rewriting the plugin script, since the supported Jython programming language of plugin can describe as complex an algorithm as Java.

The subject has successfully prototyped the web-based fault injection framework. The only technical supports we made are explaining the EFI controller interface and occasionally providing design feedback on the extensions that the subject was making. The subject has also successfully conducted a planned fault injection experiment. Including the development time of the web-based fault injection framework, the subject has spent less than two months on it, spending a majority of that time on other course work. We believe that this case study clearly demonstrates the extensibility of the EFI controller.

4.8. Summary

We have investigated the variations in the fault and error sensitivities between multiple computer systems. Our data show that there are significant similarities and differences between different types of computer systems, between different versions of the same type of computer systems, and between different types of data in the same computer system. That shows the importance of architecture and software designs in managing the fault and error sensitivities of computer systems. In particular, it shows that customized protection of different types of program data can better optimize the design trade-off between cost and fault tolerance coverage. However, we have observed that the error sensitivity of different dynamic memory objects has relatively large variations over, for example, the program execution time. That characteristic makes it difficult to statically select a set of memory objects for protection. As an alternative approach, we have presented a strategy for recoverability-driven protection that uses the recoverability of memory objects, rather than error sensitivity alone, to select the protection targets.

Chapter 5.
HAUBERK: Customized, Embedded Error Checking for Computational Accelerators

High performance and relatively low cost of GPU-based platforms provide an attractive alternative for general-purpose high-performance computing (HPC). However, the emerging HPC applications have usually stricter output correctness requirements than typical GPU applications (i.e., 3D graphics). This chapter analyzes the error resiliency of GPGPU platforms using a fault injection tool we have developed for commodity GPU devices. On average, 16-33% of injected faults cause silent data corruption (SDC) errors in the HPC programs executing on GPU. This SDC ratio is significantly higher than that measured in CPU programs (<2.3%).

This chapter then presents embedded error checking techniques for computational accelerators (e.g., GPUs). The presented framework consists of a source-to-source translator and a set of user libraries. The translator strategically places customized error detection and recovery codes in the source code of a target program so as to minimize performance impact and error propagation, and maximize recoverability. The presented technique is deployed in seven GPGPU benchmark programs and evaluated using a fault injection. The results show a high average error detection coverage (~87%) with a small performance overhead (~15%). Moreover, we present an isolated execution and deferred checking model for fault tolerance in GPUs. The presented model realizes early detection and local recovery of errors and failures of GPUs. Thus, a tolerated error or failure of accelerator is seen as a performance jitter for the rest of parallel or distributed program threads on the same node and all other nodes but does not need any global error recovery operation. We use GPU device as an example accelerator by considering its wide adoptions in state-of-the-art supercomputers and certain types of clouds.[23]

[23]This chapter was published: K. S. Yim, C. Pham, M. Saleheen, Z. Kalbarczyk, and R. K. Iyer, "Hauberk: Lightweight Silent Data Corruption Error Detector for GPGPU," in *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*, pp. 287-300, 2011.

5.1. Motivation

Graphics processing units (GPUs) are surfacing as a compelling platform for processing general-purpose HPC programs. HPC programs typically process large volumes of data using many collaborating computation tasks (e.g., science simulation or medical data processing). Modern GPUs are effective at processing such large volumes of data because of their uses of multiple cores, wide memory bandwidth, large-size register files and on-chip memory, and many arithmetic units. These rich hardware resources lessen structural hazards, and the throughput-driven design of GPU core architecture addresses both data and control hazards (i.e., main hurdles at exploiting instruction-level parallelisms in CPU designs). Furthermore, GPU hardware resources are directly exposed to the programmer by the programming model (e.g., CUDA, OpenCL, Brook+[24]).

HPC programs have strong output correctness requirements. This is in contrast to graphics programs where errors in computing colors of a few pixels may go unnoticed. Many HPC programs have quantifiable correctness requirements for their outputs. For example, in an HPC program computing a correlation function, more than 1% of value errors in any of the program output elements (e.g., a floating point number) compared with that of a golden run is treated as a silent data corruption (SDC) error. In this dissertation, an SDC is defined as an undetected data error in program output that violates correctness requirement of the program. SDC errors are serious problem in many HPC programs because of their long execution times and resulting high likelihood of experiencing hardware faults.

GPU devices targeting graphics applications usually do not need strong fault-tolerance techniques, e.g., these devices do not have any error correcting codes for memory protection [SLS06]. As a result, a relatively high hardware fault rate was observed in such devices. For example, evaluation of commodity GPU devices found at least one permanent fault in 1.8% devices [SES+09] and transient memory fault in 66% of evaluated GPUs [HP10]. Note that these transient errors are due to soft errors and/or software bugs in GPU device drivers. Recent versions of GPUs for HPC applications support memory fault tolerance techniques (e.g., CRC in GDDR5 or SEC-DED ECC). This is an important step building dependable GPU platforms for HPC domain.

Regardless of added memory error protection, HPC programs are still vulnerable to certain types of GPU hardware faults. For example, it is hard to detect faults in a GPU core (e.g., ALU, FPU, or register file) due to the irregularity and high operational speed of

[24] Brook+, http://sourceforge.net/projects/brookplus/

GPU core logic (i.e., constitutes a large portion of the silicon area in the GPU chip). Furthermore, the high-density of transistors on the die increases the likelihood of multi-bit errors [BSS08], and the integration of cores and memories contributes to an increase in hardware fault rate especially for intermittent fault [Con03].

Designing a technique to tolerate faults in GPU cores is challenging especially for HPC GPU programs because of their strong performance and cost requirements. The success of HPC applications on GPU platforms depends on the achieved computation efficiency in terms of performance versus cost or performance versus energy consumption (i.e., including the overhead for fault tolerance). From this perspective, the HPC GPU program opens a new design space different from traditional fault tolerance (e.g., for mission critical systems). Software-implemented error detection can provide lightweight cost-effective solution for this design space.

In this context, software-implemented full duplication (i.e., well-known techniques) can be an effective approach to detect SDC errors in GPU platforms. However, duplication usually doubles the program execution time. A naïve full duplication simply executes the same GPU kernel twice and compares the results from the two executions. Note that a GPU program consists of CPU- and GPU-side codes, and a GPU kernel is a part of the GPU-side code with an entry function callable from the CPU-side code. Considering the fact GPU kernels form a majority of total program execution time, the doubled execution time of GPU kernels can easily break the performance requirement of HPC GPU programs.

Optimizing naïve full duplication has achieved a limited success in GPU programs. Two optimization techniques have been studied by exploiting underutilized data- and thread-level parallelisms. The reported performance overhead is more than 84% [DMZ09]. This overhead is much higher as compared with similar techniques employed for CPU programs (e.g., [RCV+05] reports average overhead of 41%). This is because GPU programs are typically already heavily optimized and consume most of useable parallelism and computing resources in GPU.

In this chapter, we present *HAUBERK*, a software framework to derive lightweight error detection and recovery customized for target GPU programs. The derived error detection and recovery codes are strategically placed and customized by considering the performance and error propagation characteristics of the target programs. The main contributions of this chapter can be summarized as follows:

- A mutation-based, software-implemented fault injector for evaluation of commodity GPU devices.
- Characterization of the sensitivity of GPGPU applications to SDC errors. Our fault injection experiments show that single event upsets (SEU) (emulated by injection

of single-bit errors) can seriously harm reliability and data integrity of GPU kernels. For example, 18-45% of data faults cause SDC errors in evaluated GPU programs. This SDC ratio is significantly higher than that measured in CPU OS programs (<2.3%). This shows the importance of detecting and tolerating errors in GPU programs.

- Design and evaluation of two types of error detectors: (i) *duplication and checksums* to protect non-loop GPU kernel codes and (ii) *accumulation-based value range checking* to protect loop portions of GPU kernels. Our profiling results indicate that loops form a majority of GPU kernel execution time (>98% in 5 out of 7 benchmark programs).
- Design of a guardian program which re-executes GPU program in order to tolerate errors and to identify false alarms. Our error detectors detect the errors and failures of GPUs before they propagate to the CPU-side state of the same thread or states of any other thread and locally tolerate the detected errors and failures without relying on any global error recovery technique. Such local fault tolerance capability is useful to improve the scalability of parallel and distributed computer systems where a program runs multiple nodes and each node uses multiple GPUs.
- Evaluation of the HAUBERK approach on seven HPC GPU programs. The evaluation results show that the average performance overhead is 15.3% (83% reduction as compared with an optimized full duplication) and the average error detection coverage is 86.8% for injected faults.

5.2. Measurement

This section evaluates the error sensitivity of HPC and graphics programs executing on GPU and performance characteristics of the used HPC GPU programs. The parboil benchmark suite[25] is used as the source of HPC programs (six are floating-point programs and one is an integer program). Two applications (e.g., ray-trace and ocean-flow simulation) from an NVIDIA CUDA SDK (Software Development Kit) are used as 3D graphics programs.

[25] The Parboil Benchmark, http://impact.crhc.illinois.edu

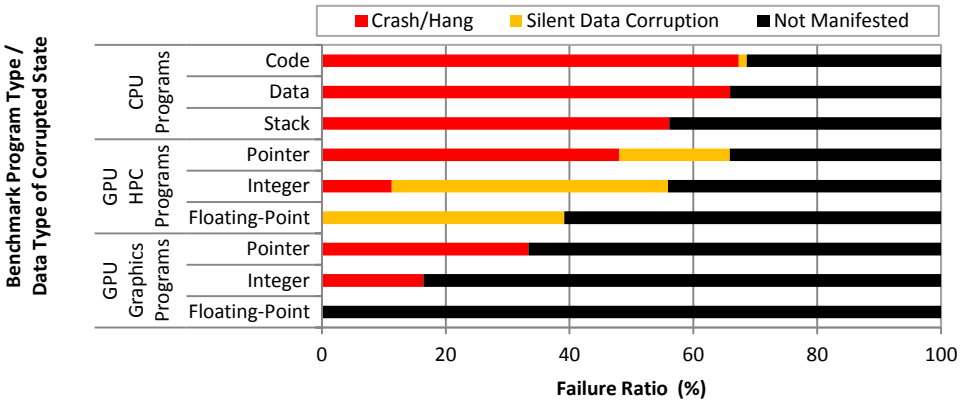


Figure 5.1. Comparison of error sensitivity of HPC GPU, graphics GPU, and CPU programs.

5.2.1. Error Sensitivity

We characterize the sensitivity of GPU programs to SDC errors. Our fault injection experiments show that SEU that is emulated by injection of single-bit errors can seriously harm the reliability and data integrity of GPU kernels.

Figure 5.1 shows the error sensitivity of HPC GPU programs, graphics GPU programs, and OS software on CPUs. The GPU program state is classified into three data types (e.g., pointer, integer, and FP data) based on the type of data where faults are injected. We inject a single-bit error into each variable in benchmark program by using the fault injection tool described in Section 5.8.

Observation 5.1. *In GPU devices, an SEU (or single-bit error) in the pointer, integer, and FP data leads to an SDC error with 18%, 45%, and 39% average probability, respectively.*

The fault injection results indicate that a large portion of injected faults lead to an SDC error in the HPC GPU programs. This shows the importance of detecting SDC errors in GPGPU.

In the HPC GPU programs, the SDC error ratio (18-45%) is higher than that observed in CPU-side system software (<2.3% according to [YKI09]). On the other hand, the failure (application crash/hang) ratio in the HPC GPU programs is lower than that in CPU-side OS programs (see Figure 5.1). The observed differences have two causes.

(a) The lack of fine-grained error protection in GPUs. Unlike to modern CPUs, GPUs do not have a page-granularity memory access permission checking that can detect many

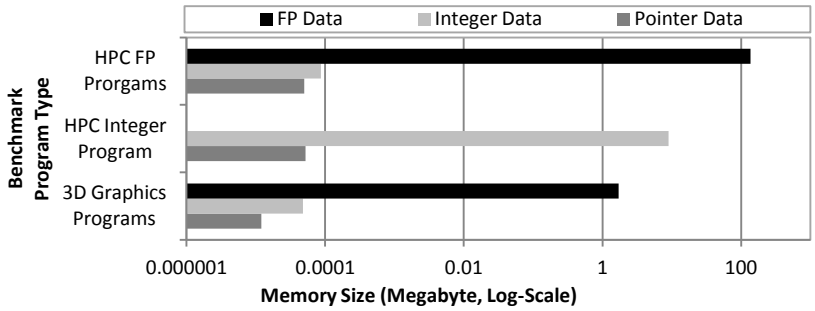


Figure 5.2. Data type vs. Memory size.

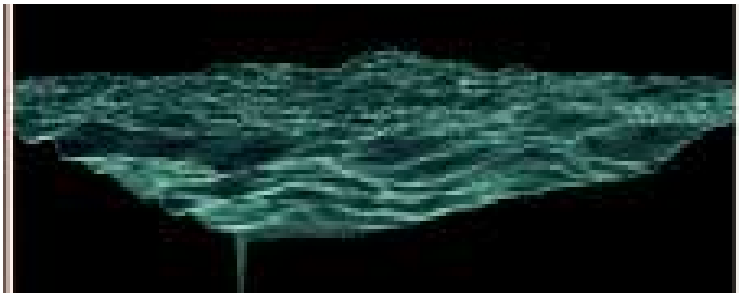
errors (e.g., corruption of a memory address). This is because of shared memory model, hardware cost, and lightweight runtime software platform in GPU devices.

(b) The massive use of FP data in HPC programs. In the benchmark HPC programs, FP data occupy 3-6 orders of magnitudes larger memory space than the pointer and integer data taken together (see Figure 5.2). Moreover, corrupted FP values are seldom detected by basic hardware protection mechanisms (e.g., divide-by-zero in FP value does not lead to an exception but returns an infinite value). On the other hand, the compared CPU programs (e.g., OS) use no FP data. When faults are injected into HPC programs on CPUs [LR04], up to 11.1% of faults in memory (e.g., heap data except for faults in communication messages) cause SDC errors, and 22% of faults in FP registers cause the same. This ratio is still lower than the SDC ratios observed in HPC GPU programs (e.g., 18%, 45%, and 39% on average). Note that such difference can be partially originated from the differences in the used benchmark programs and their output correctness requirements.

Observation 5.2. *A fault in an FP variable rarely leads to a GPU program failure, while faults (e.g., 16-33%) in pointer or integer variables are likely to cause program failures.*

In our measurements, we did not observe a GPU kernel failure due to corrupted FP value in GPUs[26]. Pointer and integer data are highly fault sensitive. This is true not only in HPC and graphics GPU programs but also in CPU programs. This is because many pointer and integer variables are used as a control data (e.g., to decide program control flow or to compute memory address). Thus, if such variables are corrupted, it can make a

[26] While perhaps rare such scenario is still possible. For example, if there is a data-flow from an FP variable to an integer or a pointer variable (e.g., FP data is used to calculate memory address), a corrupted FP value can propagate to a control data and cause a failure.



(a) Transient Fault (1 Value Error)

(b) Intermittent Fault (10,000 Value Errors)

Figure 5.3. Impact of faults in a 3D graphics program on GPU.

large drastic change in the program execution flow that is likely to be detected by basic hardware protections.

We did not notice any SDC errors caused by a single-bit error in 3D graphics programs. In graphics program, SDC error is defined as a user-noticeable corruption in video output data. This is because graphics program has a high frame rate (e.g., 30fps[27]) and a transient fault typically makes a small change in just one frame. Figure 5.3(a) shows a video frame of an ocean flow simulation program that is corrupted by a single-bit fault in its input data stream. A spike in the image is due to the injected fault.

Observation 5.3. *3D graphics programs can experience SDC errors when exposed to a longer duration fault in GPU.*

The impact of an intermittent fault having a long duration time can be significant even in 3D graphics programs. In the ocean-flow program, corruptions of 10,000 values form a prominent stripe pattern in the rendered frame image (see Figure 5.3(b)). These injected 10,000 errors emulate an intermittent fault lasting 80μs on an FPU of a 250MHz GPU with 1 instruction per cycle and 50% of execution instructions using the FPU. Note that the injected errors can also reflect impact of an intermit tent fault in a memory module or

[27] Frame per second

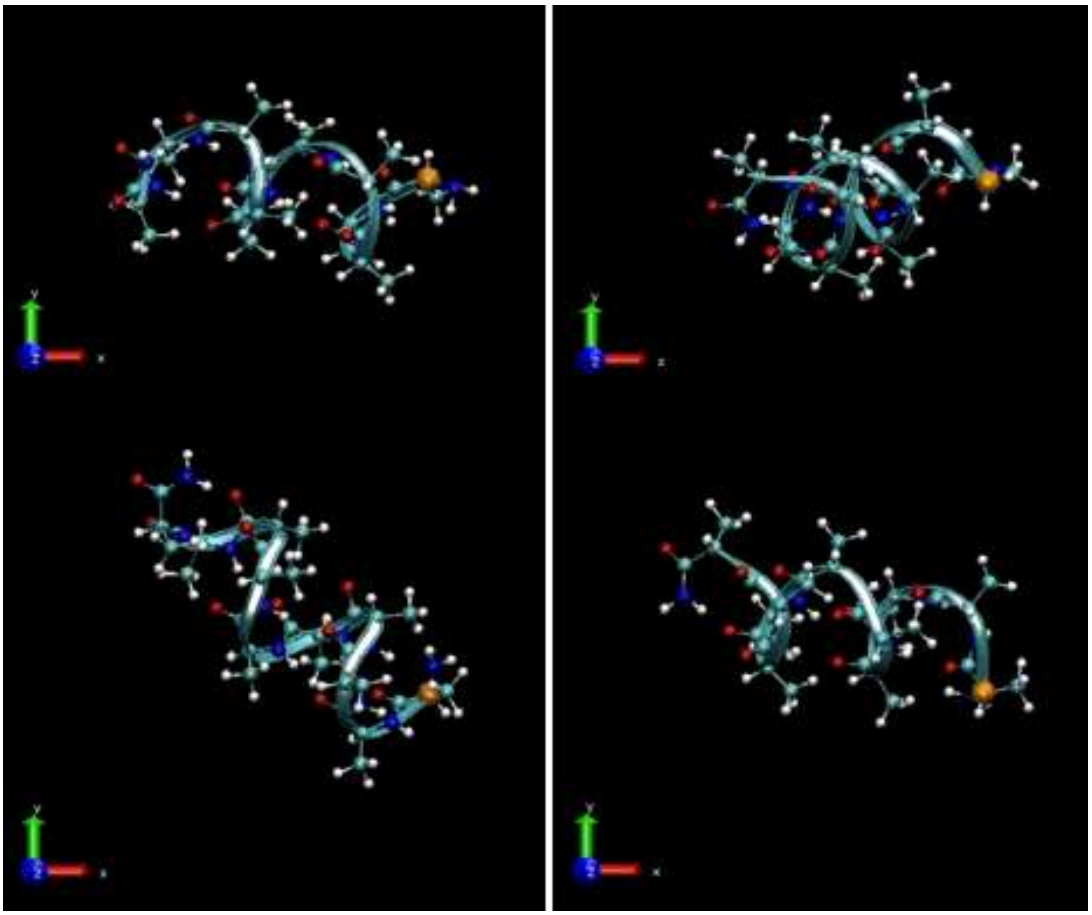


Figure 5.4. Visualization of the impact of SDCs on an n-body program.

* The intermediate simulation results of an N-body program at the same simulation time where two molecules at the top are fault-free executions and the two molecules at the bottom are faulty executions.

bus. Adding detection for SDC errors in 3D graphics programs would allow users to eliminate the corrupted frames and obtain better QoS.

Similar to the long latency failures, SDC errors are not only difficult to detect but also difficult to recover. In order to better explain the problem of SDC errors in large-scale scientific programs, we visualize the impact of SDCs on a large-scale parallel program.

We selected an *n*-body program (NAMD[28]) as the target program. This program has a large amount of randomness in terms of simulated particle positions for example. We ran the same program twice using the same parameters. Then we ran the same program twice

[28] NAMD is a scalable molecular dynamics program designed for parallel simulation of large biomolecular systems. The NAMD project is accessible at http://www.ks.uiuc.edu/Research/namd/.

again and injected a large number of faults whenever a specific type of FP operation was performed by a selected thread. Figure 5.4 shows snapshots of these four executions. The top two executions are fault-free, and the bottom two are faulty. These figures are obtained using the 3D visualization tool VMD[29].

Even for a domain expert, it was difficult to distinguish the difference between the fault-free and faulty executions if only such visual output data were given (i.e., not values that measure the internal quantities of simulated models). This is due to the non-deterministic nature of this parallel program and the visualization program. This demonstrates the difficulty and importance of detecting SDC errors using systematic techniques that check the internal states of a program and its underlying hardware.

The SDC error of an application means the corruption of its output. An SDC error of an OS kernel means there is corruption of data returned to a user process (e.g., system call output) that leads to a user process failure or to the corruption of the user process output data (i.e., application SDC error). SDC errors occur if corrupted system states are left unchecked by the baseline error detectors or if all the errors evade the checked detectors while the corrupted states propagate to the output of software.

This implies that an SDC error generally occurs when the induced errors propagate to pure program data as opposed to control data. Note that such pure data are not checked by the baseline error detectors. For example, if an error occurs in the data operand of a memory store, this modifies a value stored in the memory and causes an SDC error if the corrupted value is a part of program output. FP-type variables are typically used as pure program data in HPC programs. Due to the low error sensitivity of FP data, errors in such states are likely to lead to SDC errors. Not all corruptions in pure program data lead to SDC failures, since the definition of an SDC failure is typically application dependent. For example, some applications tolerate many errors in their output data (e.g., multimedia programs). FP programs always allow a certain degree of output value error due to the precision and accuracy problems of FPUs. If the corruption made by an error is within this allowed range, this error in the pure data does not lead to an SDC error.

5.2.2. Performance

This section characterizes the execution times of loop and non-loop portions of GPU kernels (see Figure 5.5). This data is obtained by measuring the execution time of GPU kernel with and without loops.

[29] VMD stands for visual molecular dynamics and is a molecular visualization program that displays, animates, and analyzes large bio-molecular systems by using 3D graphics and built-in scripting engines. The VMD project is accessible at http://www.ks.uiuc.edu/Research/vmd/.

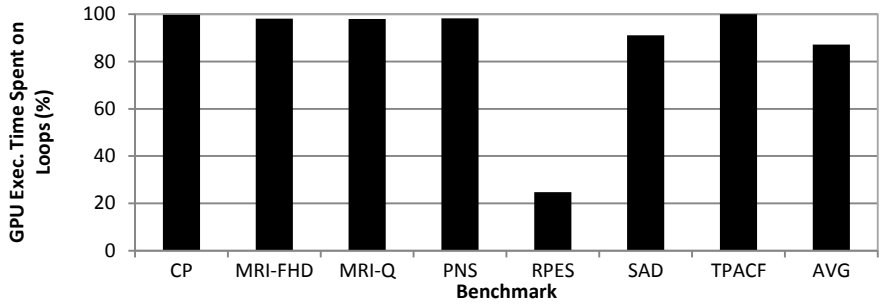


Figure 5.5. Percent of execution time on loops in HPC GPU programs.

Observation 5.4: *Loops (for, while, and do-while) form a large portion (>98% in 5 out of 7 programs and 87% on average) of the total execution time spent on GPU.*

Note that many GPU kernels are implementation of loops in original CPU codes. These loops executing on GPU typically have many iterations (e.g., proportional to the input data size) and consequently form relatively larger portions of total execution time. In contrast, non-loop codes are executing in parallel by exploiting thread-level parallelism.

The profiling data suggests that a special care is required when placing error detectors inside loop body. A small increase in the execution time of a loop can largely increase the total execution time (in accordance with Amdahl's law). For example, adding just 5 instructions inside a loop body can degrade the performance of a GPU-side code by 22% if the loop has 20 instructions and the loop forms 90% of the total GPU kernel execution time. For GPU programmers, loops are one of the main optimization targets and thus often have a small number of instructions.

5.3. Related Work

This section classifies and analyzes existing error detection techniques potentially applicable in the context of this study (see Figure 5.6). The design goal is to find a high coverage detector without compromising performance.

(i) Naïve full duplication. This basic technique has high SDC error detection ratio (close to 100%) but almost doubles the execution time. Duplication uses either temporal or spatial redundancy. Spatial redundancy is achieved by duplicating GPU hardware. The technique can quickly detect errors; however, synchronizing original hardware and its replica brings ~50% of extra performance overhead together with doubled hardware cost [SLS07]. Software technique can easily create temporal redundancy (e.g., by instrumenting the source code of a target program). R-Naïve [DMZ09] executes same GPU kernel

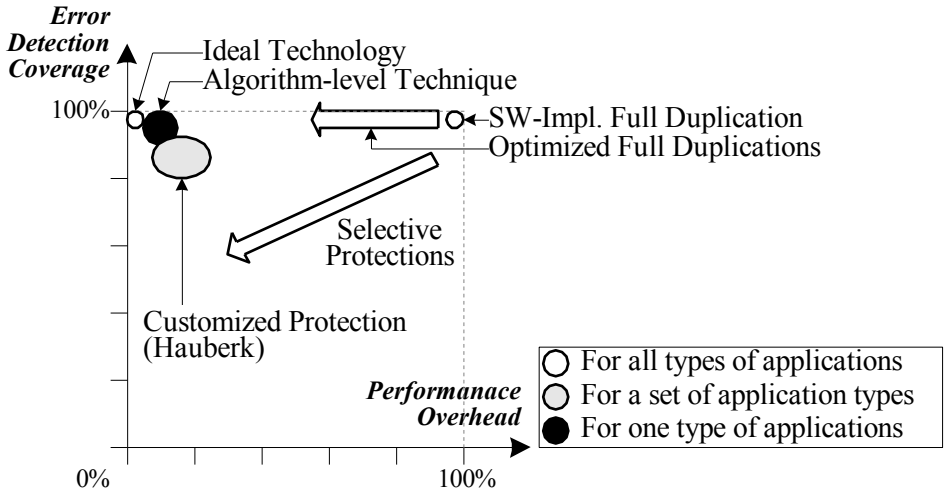


Figure 5.6. Spectrum of various types of data error detection techniques.

twice by using two different copies of memory data. R-Naïve has a good SDC error detection ratio (~100%) but it also almost doubles the GPU execution time and CPU memory space used to keep input and output data (see software-implemented full duplication in Figure 5.6). We found that real GPU failure examples (e.g., long execution time error) where these existing full duplication approaches cannot detect and tolerate (Section 5.10).

(ii) Optimized full duplication. This approach utilizes idle hardware resources for processing extra computation brought by the duplication. SWIFT [RCV+05] extends and applies an instruction duplication technique (EDDI [OSM02]) to a VLIW-type CPU processor by duplicating backward computation slices for address and data values of all memory write operations. These duplicated instructions are reordered by compiler (or hardware scheduler) before execution to exploit the instruction-level parallelism in VLIW (or superscalar) processors. On an Itanium CPU, SWIFT shows ~100% data error detection ratio with ~41% performance overhead, on average (see optimized full duplication Figure 5.6).

The study reported in [DMZ09] employs optimized full duplication and shows that the approach is not highly effective for GPU programs in the way it is for CPU programs (e.g., SWIFT). After optimizations by exploiting data- or thread-level parallelism in GPU, >84% of performance overhead is shown in widely-used GPU programs. This is because GPU programs are heavily optimized such that original program already uses most of the usable hardware resources in GPU, while the duplicated computation seeks same types of hardware resources or parallelism as the original one.

(iii) Selective protection. This approach selectively protects parts of the program state in order to reduce the amount of extra computation to detect errors. Error detectors are strategically placed in highly error sensitive state. This is motivated by fault injection results, which indicate that many faults in lower-layers of the system are masked and do not manifest in applications [WQR+04][GKI04][YKI10].

(a) Fault injection. Fault injection can be used to find error sensitive program state [HJS02]. This is most effective if the size of the program state (e.g., code and data) is small. Otherwise, it can take a long time to analyze the error sensitivity of a large-size program. The large volume of GPU program state (e.g., several gigabytes memory data and several 100 or 1,000 threads) can make the fault injection approach impractical if fine-grained (e.g., a data word) sensitivity analysis is needed.

(b) Static complier analysis. Compiler-based heuristic algorithms can quickly select protection target state even in large-size programs by using static source code analysis. For example, an early technique [PKI07] analyzes the number of possible uses of each program variable and selects a certain number of variables from one with the largest possible uses. This technique detects 41% of SDC errors with 33% performance overhead when applied to CPU programs. Another technique [FGA+10] excludes program states from the protection if errors in the state can quickly lead to the program crash.

(c) Dynamic program analysis. This derives and selects likely program invariants by profiling and monitors selected invariants at runtime. For example, if a variable always contains a value between min and max during profiling [ECG+01], this generates an error detector to check whether the value of this variable is in the identified boundaries. Because profiling uses a limited number of input data, the derived detectors may lead to false positives and that can be addressed by an on-line diagnosis [SLR+08].

(iv) Algorithm-based fault tolerance. Error detection techniques designed and optimized for a particular type of algorithm or program are usually highly efficient in terms of error detection coverage and performance overhead. For example, a technique [HA84] customized for matrix multiplication algorithms detected ~99% of SDC errors with only a small amount of overhead. A software technique [MNM10] customized for GPU global memory errors can detect memory errors with a negligible overhead in compute-intensive applications.

5.4. GPU HAUBERK

HAUBERK generates customized error detection and recovery routines for GPU programs by leveraging common performance and reliability characteristics of GPU programs.

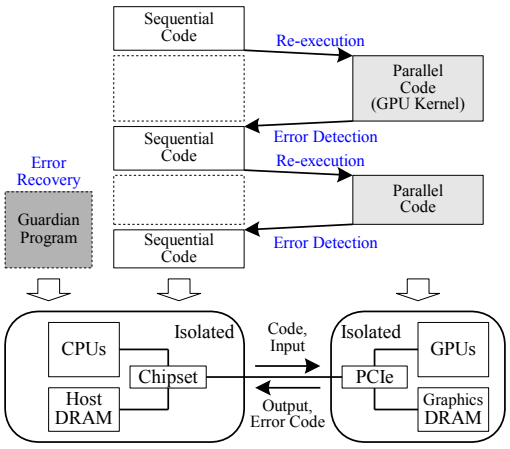


Figure 5.7. Isolated execution and deferred checking model of *HAUBERK*.

5.4.1. Design Principles

The key principles used to design error detector derivation algorithms are:

Principle 7.1: *HAUBERK customizes error detectors by using profiling information of common HPC GPU programs in order to minimize the impact on performance.*

HAUBERK uses three types of error detectors for loop portion of codes in GPU program, non-loop portions of codes, and GPU kernel output data with spatial or temporal value similarities. This is motivated by the loop execution time measurement data (see Section 5.2.2). Considering the small contribution of non-loop codes to the overall execution time, strong error detection techniques are designed for non-loop codes. On the other hand, error detection techniques for loop codes are designed and optimized to minimize the performance impact (e.g., by adding only two addition instructions inside a loop).

Principle 7.2: *HAUBERK selectively protects the program state where errors in other states are likely to propagate.*

The *HAUBERK* loop error detector selectively protects program states where computation of the state directly or indirectly uses many other variables. This means errors in these variables are likely to propagate to protected states and thus are likely to be detected by strategically placed error detectors. The detection accuracy is improved by customizing loop error detectors for common patterns in FP value distributions.

Principle 7.3: *HAUBERK places error detectors by considering the recoverability of errors (i.e., urgency in error detection to enable and support safe error recovery).*

HAUBERK defers placements of error detectors as long as possible by taking advantage of inherent hardware-enforced error isolation between GPU and CPU (i.e., provided by

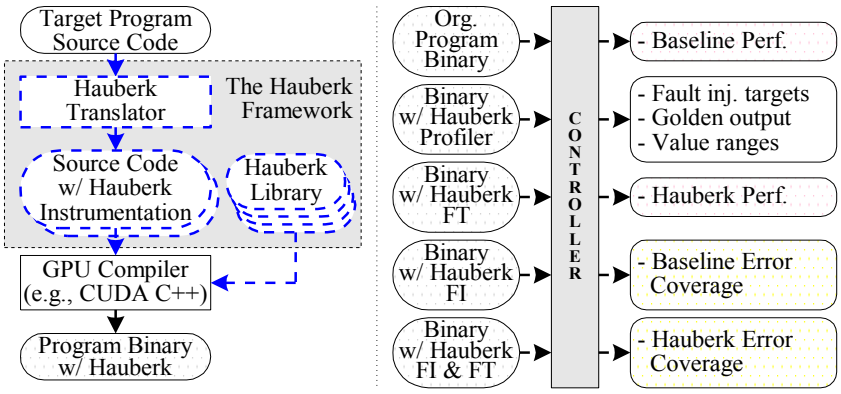


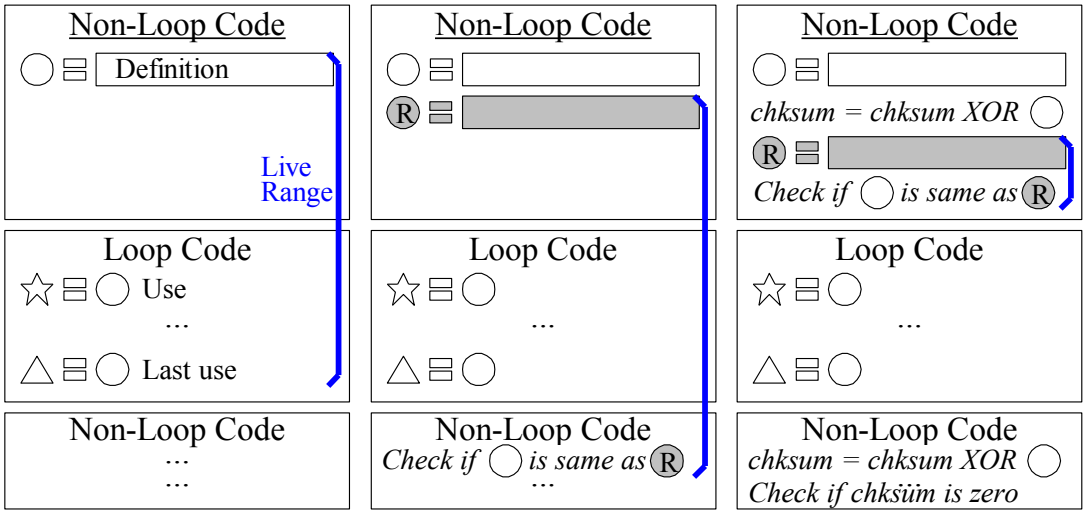
Figure 5.8. Compilation and evaluation flows in the *HAUBERK* framework.

private memory and explicit communications of GPU and CPU states, see Figure 5.7). Many errors occurring in GPU-side programs are detected by basic hardware protection before propagating and harming the availability of CPU-side control software (e.g., OS). For example, GPU runtime can detect all GPU kernel crashes by default. Thus, we focus on detecting SDC errors in the output of GPU kernels because this is a practically feasible error propagation path from GPU kernel state to CPU-side program state. Then, in order to further reduce the performance overhead, we defer the placement of error detectors in GPU kernel code as long as possible that avoids or reduces coverage overlap between the basic hardware-enforced detectors and *HAUBERK*-generated detectors.

Like any other error detector checking intermediate program state, error detectors in *HAUBERK* can suffer of false alarms (i.e., an error in intermediate program state does not propagate to final program output nor makes an observable change in the output). In order to enable a diagnosis of false alarms, we defer reporting detection of suspicious behavior until the end of a GPU kernel execution. If the kernel completes without a failure, its output data is reported to the CPU code together with the error detection result. Any reported error triggers a recovery process in the CPU codes that can identify false alarms by re-executing the GPU kernel and comparing the returned outputs.

5.4.2. Framework

HAUBERK library is a set of user-level C libraries. Each library defines a collection of variables and functions for codes added by the *HAUBERK* translator. Four types of li braries exist: profiler, FT (fault tolerance), FI (fault injector), and FI&FT. We generate a program binary file using each of these libraries (see Figure 5.8). Specifically, the original program binary is used to measure baseline performance. A program binary with the



(a) Original code (b) Naïve duplication (c) *HAUBERK-NL*

Figure 5.9. Duplication techniques for non-loop codes where statements marked as gray symbols or italic texts are added for error detection.

HAUBERK profiler profiles value ranges of variables protected by loop error detectors, derives all fault injection targets, and gets the output of the golden run. A program binary with *HAUBERK* FT is used to evaluate the performance overhead of placed *HAUBERK* error detection and recovery routines. A program binary with *HAUBERK* FI is used to analyze error detection coverage and error sensitivity of baseline program. Finally, a program with *HAUBERK* FI&FT is used to evaluate the error detection coverage of placed *HAUBERK* fault tolerance routines. We use a GUI-based controller program to automate this evaluation process when many experiments are needed (e.g., for fault injection).

5.5. Detection Technique for Non-Loop Codes

This section describes the error detector derivation algorithms for non-loop codes of GPU kernel. Note that a GPU kernel can have one or more loops with non-loop codes before, after, and between these loops. Many variables defined in non-loop codes are control data (e.g., pointers, constant input data, and data for control-flow conditions).

The definitions of all virtual variables defined in non-loop codes are duplicated in source code. In this chapter, virtual variable means a subset of the live range of program state where the subset has one definition and multiple uses.

A naïve variable-granularity duplication can duplicate the definition of virtual variable and check the original and duplicated variables after the last use (or the immediate post-

dominator of last uses) as exemplified in Figure 5.9(b). This can largely increase the register pressure (e.g., by two times) because the duplicated variable has the same live range as the original variable. Note that in the GPU, if physical registers allocated to each thread are insufficient, register spill operations occur, which slows down the performance due to memory accesses (e.g., to an on-chip cache or an off-chip DRAM).

HAUBERK-NL (namely, *duplication-and-checksum*) duplicates the definition of the virtual variable and immediately checks the original and duplicated variables (see Figure 5.9(c)). This check is done to detect errors that may occur during the computation (e.g., in ALU or FPU). To detect errors occurring after this computation (e.g., errors in register file), we update the checksum variable by XORing the original variable value to it. This checksum update is done right before the comparison operation to prevent losing errors that occurred between the comparison and the checksum update (see Figure 5.9(c)). The checksum variable is updated (i.e., XOR) once again using the original variable after its last use or the immediate post-dominator of last uses (e.g., after the loop in Figure 5.9(c)). The checksum variable is 4 bytes. If a variable size is not 4 bytes, it is aligned by four-bytes for XOR operations. This checksum variable is shared for all duplicated virtual variables in same kernel (i.e., even in nested functions). This variable still shall be zero at the kernel exit because it is XORed twice with each and every duplicated virtual variable value.

This checksum-based duplication avoids a large increase in the register pressure. Only one checksum variable is added per kernel because one checksum variable is used for multiple virtual variables. The duplicated variables are alive only for two statements (i.e., one for its definition and the other for checking). The increase in the live range of the original variable in the presented technique (e.g., from last uses to the immediate post-dominator if there are multiple last uses) is same as that of the naïve duplication technique. Register pressure control in this duplication and checksum technique efficiently leverages a common characteristic in GPU architecture that memory operations are more expensive than computation operations.

The derivation algorithm of non-loop error detectors has five steps:

(i) Update checksum. After the definition of each virtual variable in non-loop codes, the algorithm inserts a statement to update the checksum by using the defined virtual variable value. *(ii) Duplicate computation*. This step duplicates the definition statement of the target virtual variable. Another variable (i.e., temporally allocated in a register) is used to keep the duplicated computation result. *(iii) Check computation result*. This step inserts an *if*-statement to compare the original and duplicated virtual variable values. The live range of this duplicated variable ends here. Although this *if*-statement is a point of control-flow

divergence, because all threads in a same warp (i.e., unit of thread scheduling in GPU) make the same control-flow decision if there is no fault, this does not introduce a large performance or scheduling overhead. *(iv) Update checksum*. This step inserts an XOR statement to update the checksum variable again by using the original virtual variable. The inserted location depends on the number of uses of the original virtual variable. For example, if the variable is used but not updated inside a loop, this algorithm inserts an XOR statement after the loop. If the variable is updated inside a loop, this XOR statement is inserted right before the loop (i.e., introducing an uncovered window). Note that variables updated inside loops are protected by error detectors for loops described in Section 5.6. *(v) Validate checksum*. This step inserts an *if*-statement as the last statement of the GPU kernel to check whether the checksum is zero. A statement is added to set an SDC error bit at runtime if the checksum is non-zero.

The described algorithm repeats the steps from (i) to (iv) for all virtual variables defined outside of the loops. In the case of function parameters, the checksum is updated only (i.e., without duplication) at the entry and exit of the kernel function if the parameter is not modified inside the kernel. This can detect corruptions in parameters. If a parameter is updated inside the function, its second checksum update is done before the update statement, and the updated parameter is treated as another virtual variable (i.e., protected separately). The same derivation rule applies to memory load expressions and statements.

The delivery of potential error detection report from GPU to CPU is done by using an object in memory (namely, control block). CPU-side program allocates a control block in its memory, copies the allocated object to GPU memory, and delivers the pointer of copied object as a parameter of GPU kernel. Placed error detectors (i.e., added *if*-statement) use this passed control block and marks detection results. If the GPU kernel completes normally, the CPU-side code copies this control block back to CPU memory and tosses it to the error recovery engine described in Section 5.7. This control block also delivers other information between CPU and GPU (e.g., to configure loop error detectors) as described in Table 5.1.

5.6. Detection Technique for Loop Codes

This section describes the error detector derivation algorithms for loop codes of GPU kernel. Many variables manipulated inside loops are streamed input and output data.

We present *value-accumulation-based range checking* for loop codes. Derivation of this error detector has four steps:

Table 5.1. Descriptions of Instrumentations Used for *HAUBERK*.

<table>
<tr><th>Location / Lib.</th><th>FI (Section VII)</th><th>Profiler (Section V.B)</th><th>FT (Section V.A., V.B., VI)</th></tr>
<tr><td>[CPU] Top of the main file</td><td colspan="3">Includes a header file for HAUBERK libraries</td></tr>
<tr><td rowspan="2">[CPU] Entry of main()</td><td colspan="3">Initializes the control block</td></tr>
<tr><td>The control block is for the location, time, and type of fault injection target</td><td>The control block is for profiled value ranges and execution counts</td><td>The control block is for value ranges, detection results, and outliers</td></tr>
<tr><td>[CPU] Exit of main()</td><td>Stores fault activation result to a file</td><td>Stores profiling results to a file</td><td>Stores updated value ranges to a file</td></tr>
<tr><td rowspan="2">[CPU] Before launching GPU kernel</td><td colspan="3">Copies the control block from CPU to GPU</td></tr>
<tr><td colspan="2">-</td><td>Notifies this to guardian process and calls a checkpoint library (option)</td></tr>
<tr><td rowspan="2">[CPU] After GPU kernel launch</td><td colspan="3">Waits until the kernel completion and copies the control block back from GPU to CPU</td></tr>
<tr><td colspan="2">-</td><td>Calls an error recovery function</td></tr>
<tr><td>[CPU] GPU kernel function</td><td colspan="3">Adds a pointer variable for the control block as a function parameter in GPU kernel function prototype and its caller(s)</td></tr>
<tr><td rowspan="2">[GPU] After definition of virtual variable in GPU non-loop</td><td colspan="2">Calls a library function with an identifier, pointer, type, and used hardware components of variable defined in previous statement</td><td rowspan="2">Updates a checksum variable, duplicates the definition, and checks original and duplicated variables</td></tr>
<tr><td>To inject a fault into a defined variable at a designated time of execution</td><td>To count execution count per variable</td></tr>
<tr><td>[GPU] After def. of virtual variable in GPU loop</td><td>Same as “After definition of virtual variable in GPU non-loop” field</td><td colspan="2">Adds two addition statements for each protected target virtual variable (one for target variable and the other for counter) and merges the counters if possible</td></tr>
<tr><td rowspan="2">[GPU] Before loop in GPU kernel</td><td rowspan="2">-</td><td colspan="2">Defines accumulator and counter variables for each protected loop variable</td></tr>
<tr><td>-</td><td>Updates the checksum var. if needed</td></tr>
<tr><td>[GPU] After loop in GPU kernel</td><td>-</td><td>Profiles value ranges of accumulated variables divided by their counter</td><td>Checks accumulated variable value ranges and updates the checksum var.</td></tr>
<tr><td>[GPU] Exit of GPU kernel</td><td>-</td><td>-</td><td>Checks the checksum variable</td></tr>
</table>

(i) Select target variable for protection. Among all virtual variables defined inside a target loop, we first select self-accumulating virtual variables. This is because these variables do not need any extra code added inside the loop for protection. We then exclude virtual variables that have forward dataflow dependency to these selected variables from the dataflow graph of all virtual variables inside the loop.

Among the reminder of virtual variables, we select a virtual variable with the largest cumulative backward dataflow dependency. As shown in Figure 5.10, a cumulative backward dataflow dependency means the number of virtual variables defined inside a loop and unprotected by non-loop error detectors that can directly or indirectly be used to com-

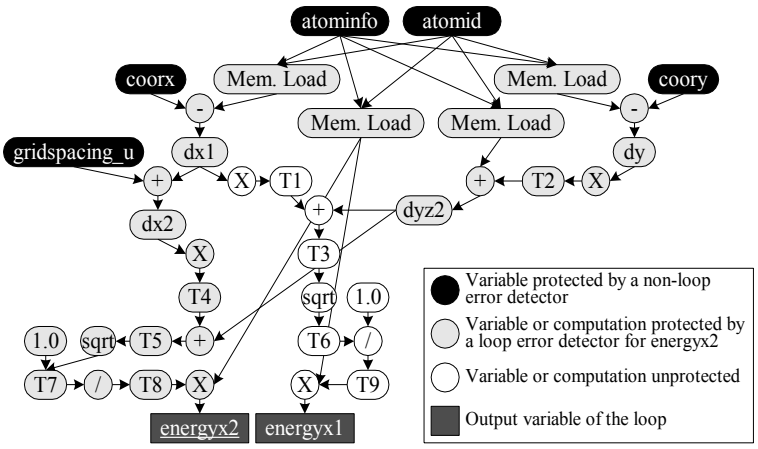


Figure 5.10. Dataflow graph of a loop in a coulombic potential GPU kernel.

pute the target virtual variable. Thus, a larger cumulative backward dataflow dependency means a higher chance of propagation of errors in other system states to the target virtual variable. If a technique is available that can detect even small corruption in the target variable, this can cover errors in the program state by only checking a few variables.

Figure 5.10 exemplifies a data-flow graph of a loop in a GPU kernel that is computing a coulomb potential function. A circle means a binary or unary operator, and both box and ellipse mean either a virtual variable or a temporary variable where the name of temporary variable starts with *T*. A temporary variable is used for virtual variable defined by using multiple binary or unary operations, and each operation has an intermediate program state in register or memory. In this example, two output variables (i.e., either live after the loop or written to memory) exist that are marked as black boxes. The cumulative backward dataflow dependency of *energyx1* and *energyx2* are 12 and 13, respectively, including the memory load data but not the constant (i.e., 1.0 in the figure). Here, we exclude five virtual variables that are not modified inside the loop and are protected by non-loop error detectors (i.e., black ellipses in the figure). Thus, we first select *energyx2* for protection.

Users can specify the maximum number of virtual variables (Max_{var}) that can be protected by these loop error detectors. Note that Max_{var} counts self-accumulating variables. If Max_{var} is higher than one, this selection is repeated Max_{var} times. Before repeating this selection process, we remove the previously selected virtual variable(s) and other virtual variable(s) having forward data dependency to the previously selected ones from the dataflow graph. This is to select and protect another virtual variable that can cover the largest number of previously unprotected (either directly or indirectly) virtual variables. Note that this repetition eventually terminates because there is only a finite number of virtual variables in any loop.

(ii) Generate value accumulator code. The placed error detector accumulates the data value of each protected virtual variable in every loop iteration. This step is skipped if a protected variable is a self-accumulator. For each protected virtual variable, another variable is defined with the initial value of zero (e.g., *float accumulator = 0.0;*) right before the loop. Using this accumulator variable, an accumulation statement is added right after the definition of the protected virtual variable (e.g., *accumulator += energyx2;* if the protected variable name is *energyx2*) inside the loop.

(iii) Generate accumulation counter code. An addition statement is added to count the number of accumulation operations for each accumulator variable. The *HAUBERK-L* translator defines an integer variable right before the loop (e.g., *int iterator = 0;*) and adds an integer addition statement (e.g., *iterator++;*) inside the loop right after the placed accumulator(s). Even when many accumulator variables are used, these variables often have an identical control-flow path (e.g., common case is accumulation count is same as the loop iteration count). In this case, these variables can share one accumulation counter. If the accumulation count is expected to be same as the loop iteration count, we maintain this custom accumulation counter because this is also used to detect some loop control-flow errors (i.e., errors in loop iterator, termination condition, or iterator manipulation operation).

(iv) Generate error checking code. An error checking routine is added right after the loop code. This added routine calls a function defined in the *HAUBERK* FT library (i.e., *HauberkCheckRange(...)*) by using the averaged accumulator value (e.g., *accumulator*/*iterator*) and the pointer to control block. The called function checks whether the current accumulation value is within the profiled value ranges (i.e., specified in the control block). If the value is outside of ranges, this function calculates new ranges (i.e., assuming it is a false positive) and stores this to control block together with setting an SDC error bit. The updated ranges are used by the recovery engine as a part of its on-line learning process. In the FT library, the function called at the entry of *main()* loads the profiled value range from a file and configures the control block for loop error detectors. Another function called at the exit of *main()* stores the updated value ranges to the same file if false alarm is detected.

The detection code for an example in Figure 5.10 is as follows where bold texts are added for *HAUBERK-L* protection:

```
float  accumulator = 0;  int  iterator = 0;
for (atomid=0; atomid < numatoms; atomid++) {
    …
    accumulator += energyx2;
    iterator++;
```

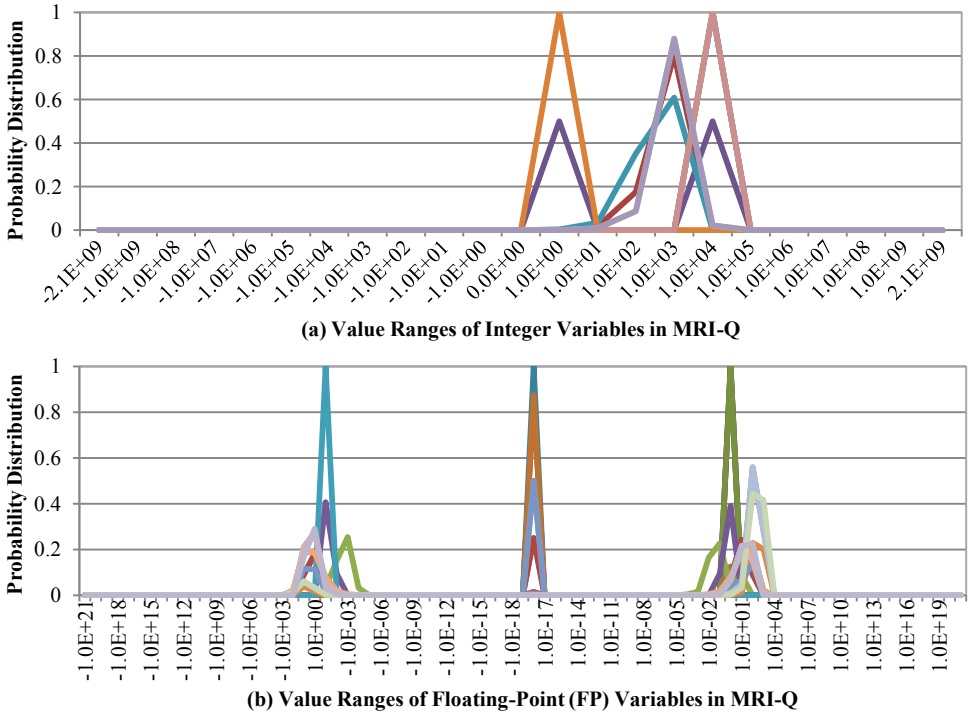


Figure 5.11. Value range distributions of integer (a) and FP (b) variables in the *MRI-Q* program executing on a GPU device.

```
}
HauberkCheckRange(controlblock, 0, accumulator / iterator);
HauberkCheckEqual(controlblock, 0, iterator, numatoms);
```

It also has an added code after the loop to check the loop iteration count (i.e., *HauberkCheckEqual(...)*). Often, we can calculate the loop iteration count (e.g., loop iteration count is *MAX* for a loop, i.e., *for(int i=0; i<MAX; i++)* { … }). This loop iteration count is treated and checked as an invariant of the program. Even if the calculation of the loop iteration count is complex (e.g., using two conditions), we find that it is still feasible in many cases to drive a statement that can dynamically calculate the iteration count. For example, for a loop *for(int x=0, y=0; x<A && y<B; x++, y++)* { … }, the loop iteration count is same as the minimum of *A* and *B*. Also if a condition variable can be changed inside the loop, the iteration count is computed and stored in a variable before the loop.

If a hardware fault makes a large change in the averaged accumulated value, this is likely to be detected by this value range checking. On the other hand, if an error makes only a small change in the value of protected variable, this will not be detected by the checking as far as the corrupted value is within the checked value ranges. Note that this

can also be a case that the error also did not significantly impact the program output so to cause an SDC error.

The use of value range checking in GPU programs is motivated by our measurement data. A strong correlation is observed in values stored in or computed for a same program variable in many HPC GPU programs. This strong correlation is observed in both integer and FP data (see Figure 5.11). Figure 5.11 shows the value distribution of integer and FP variables in an HPC GPU program (*MRI-Q*). Each graph line represents the value distribution of a single variable, and *x-axis* means integer (or FP) numbers that can be encoded by 32-bit integer (or FP) variable where *1.0E+N* means 1^N and *1.0E-N* means 1^{-N}. Integer values computed by the same code fragments are likely to be in adjacent two units of power of 10s. Most of these graph lines have a sharp peak higher than >0.5. This means that >50% of values computed for the same variable are likely to be in a single unit of power of 10s. Similar characteristics are observed in other HPC GPU programs.

Note that variables in the same program have relatively similar correlation points in both FP and integer data. This is because these variables have direct or indirect data flows to each other and thus their values are correlated.

An important finding is that many FP variables have three correlation points. Two correlation points are in negative and positive numbers with a similar magnitude, and the other point is in close to zero. Values in each correlation point are strongly correlated to each other (e.g., most of correlation values have same order of magnitude). Considering the wide value space that an FP variable can encode (e.g., approximately $2^{-126} \sim 2^{128}$ for single-precision positive FP numbers), a typical FP program uses a small fraction of the available FP value space, making this value range checking effective in FP data.

Based on this finding, the value range profiling algorithm is specifically designed to detect up to three correlation points. We set two default threshold points (e.g., at -10^{-5} and 10^{-5}) and treat any value observed between these points as a value correlated to the correlation point at zero. Other values outside of these threshold points are correlated to the correlation point in positive or negative numbers. We sum up the sizes of value spaces of all value ranges identified by this profiling. We then change the two threshold points (i.e., by multiplying either 10 or 0.1 to examine its neighbors) and repeat the same profiling. This process is repeated if the calculated total value space is smaller than that measured in the previous run.

If a potential SDC error is detected, this error detector does not terminate the GPU kernel. This instead defers error reporting until the kernel completion. If the kernel causes a failure, it validates that this detection is not a false alarm.

Places where *HAUBERK* translator adds or mutates source codes are summarized in Table 5.1. It shows the exact changes made by the *HAUBERK* translator depending on the type of used library.

5.7. Recovery Technique

The detected errors are not only diagnosed but also tolerated by the presented selective re-execution technique.

HAUBERK-L error detectors may result in both false positives and negatives. A *false positive* occurs when a new input data produces a value for accumulator variable that is not in the profiled value ranges. This is because used value ranges are derived by profiling that only uses a limited number of input samples. Especially when *HAUBERK-L* is used, using many representative samples in profiling can reduce the likelihood of false positives but it cannot guarantee complete removal of false positives in many real world applications. A *false negative* occurs for example when the averaged accumulated value of *HAUBERK-L* is within the profiled value ranges after the program experiences a fault, while the program output is largely corrupted and violates its correctness requirement.

False SDC detection alarms are identified by re-execution. When loop error detector reports a potential SDC error, the recovery assumes this is a false positive and re-executes the GPU kernel for diagnosis (see Figure 5.12).

(i) False alarm. If the re-execution also raises an SDC alarm and its output is identical to the original output, these two are likely to be false alarms (i.e., false positive). Here, *identical* means each value in the output of one execution is the same as the corresponding value of the output of the other execution if the output of GPU program is always deterministic. If a nondeterministic GPU program is used, output values showing a certain degree of difference (i.e., more than twice of the output correctness requirement – a conservative approach is used because the golden run output is not available) are still treated as identical. Up on a detection of a false positive, *HAUBERK-L* technique stores the updated value ranges to a file (i.e., a part of on-line learning process).

(ii) SDC error due to transient or short intermittent fault. If the re-execution terminates normally and does not raise an SDC alarm, we assume that the alarm raised in the first execution is due to transient or intermittent fault (i.e., removed before the second execution). In this case, the re-execution result is taken.

(iii) SDC error due to long intermittent or permanent fault. If the re-execution also raises an SDC alarm but its output is not identical to the original execution output, we ex-

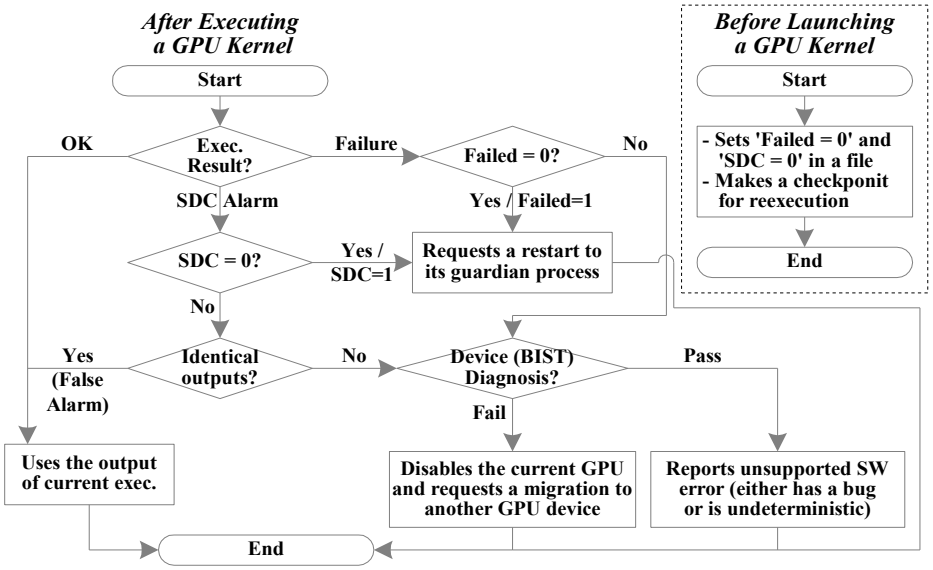


Figure 5.12. Error diagnosis and recovery algorithm.

ecute a GPU program that is specifically designed to produce multiple sets of output data by examining various parts of GPU hardware. The functionality of this program is similar to built-in self-test (BIST). If this program detects a hardware fault, the current GPU device is disabled and another device in the node or cluster is used for re-executing the current GPU program. A daemon process is periodically running this program on disabled GPU devices with a time delay ($T_{backoff}$). Here, $T_{backoff}$ is doubled after every execution of this program. If the error was due to an intermittent fault, this configuration reduces the utilization of GPU device, and once the fault is removed, this program can re-enable the GPU device.

(iv) Configuring error detectors. This false alarm diagnosis can calculate the false positive ratio. For example, if the current false positive ratio of a *HAUBERK-L* is higher than a threshold (e.g., 10%), the recovery engine increases the parameter *alpha* (e.g., by multiplying 10) for the error detector (see Section 5.6). If the false positive ratio is smaller than another threshold (e.g., 5%), it reduces the *alpha* (e.g., divides by 10) as far as *alpha* is larger than or equal to 1. Specifically, the maximum value of each value range of *HAUBERK-L* is multiplied by *alpha*, and the minimum value of each value range is divided by *alpha* if these maximum and minimum values are positive numbers. The use of loose value ranges can reduce false positives but at the same time can increase false negatives. This tradeoff between false positives (i.e., performance overhead due to re-execution) and false negatives (i.e., detection accuracy) is analyzed in Section 5.10.3.

5.8. Dependability Evaluation Framework

We develop another SWIFI for commodity GPUs. In order to analyze and evaluate the dependability of GPUs, a dependability benchmarking tool is required. However, there is no published fault injection tool that can be used to perform experiments on programs running on real GPU devices (e.g., only a simulated fault injector is publically known [SLS06]). We build a SWIFI toolset to emulate single- and multi-bit soft errors in GPU processor and memory states. This source code mutation-based fault injector does not require any modification in GPU hardware and hence is applicable to commodity GPUs.

A source code mutation (i.e., embedding error injection code) technique is used to efficiently control fault injection target (i.e., a state of a thread on one of several 100s GPU cores). Although our current implementation is done for CUDA, this tool can be easily ported to other parallel programming languages (e.g., OpenCL). Note that we use this mutation-based fault injection in GPU because many GPUs do not have suitable breakpoint mechanism (e.g., either lack of such hardware feature or only supporting a relatively inaccurate breakpoint feature partially due to the offloaded nature of GPU threads from its host control software on CPUs). This source code mutation-based injection is also applicable for other commodity processors that have neither a hardware- nor software-breakpoint feature because this mutation-based method needs no hardware modification.

The *translator* component in the control node embeds the fault injection code in the source code of target GPU program. The compiled binary of the translated source code is transferred to an injector node. When the copied binary runs, a fault is automatically injected into a targeted state of the GPU program at runtime, and error and failure information are printed to the console. The standard output of the GPU program is captured and processed by the *result interpreter* of a monitor node to produce a fault injection result file.

The embedded fault injection code is to select both a fault injection target (e.g., a variable of a thread) and an injection time (e.g., *j*-th loop iteration of *i*-th call of a function). In the current implementation, a function call statement is added after each program assignment statement (see gray boxes in Figure 5.13). This is to call a library function that actually injects an error. For each GPU kernel statement that can change a program state, the source-to-source translator component derives the symbol name and data type of a variable (or a program state) that can be changed as a result of the statement execution. A func-

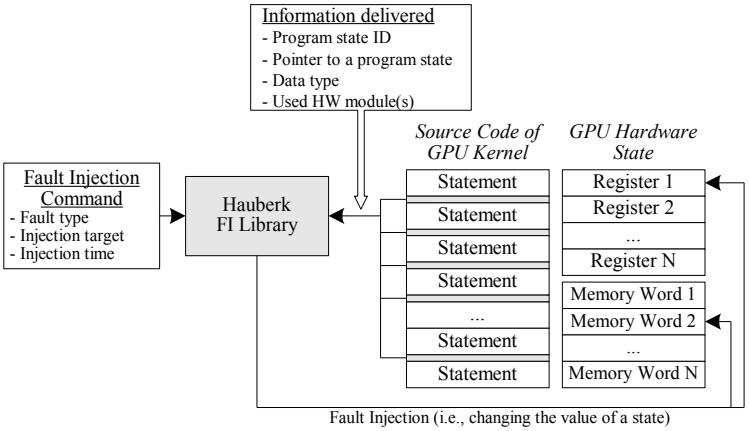


Figure 5.13. A GPU kernel with mutation-based fault injection codes.

tion call statement is added after each program statement[30] to call a fault injector library function (see Figure 5.13). Fault injection information is delivered as arguments of this function call. The arguments are: the identifier of the fault injection target variable, pointer to the variable, identifier of the data type corresponding to the target variable, and hardware components used by the preceding statement. Used hardware components are statically derived by analyzing the types of used operations (e.g., ALU for integer and FPU for FP operation). The provided fault injection library changes the value stored in the derived variable as specified by the fault type. If the derived variable is for an FPU register, this register value is copied to an ALU register. This is because the fault injection uses logical operations (e.g., XOR) that are only supported by ALU in some GPUs. The changed value in the ALU register is then copied back to the original FPU register.

Hardware faults can occur in any transistor (or component) in GPU. A fault is either masked (e.g., when the faulty transistor is not in use) or it propagates to a micro-architectural state. Similarly, a fault in a micro-architectural state can propagate to upper-layers (e.g., a software-visible architecture state). This mutation-based injector emulates errors in software-visible architecture states to evaluate the impacts of faults on the relia-

[30] Although not common, adding many call statements in the source code of GPU kernel can cause a GPU runtime error if used GPU device does not have sufficient hardware resources. In this case, fault injection target is selected at compile-time. Specifically, the variable identifier of a fault injection target is given as input of the translator that adds only one call statement in GPU kernel source code where the added call statement has the given variable identifier as its parameter. This, however, increases the total fault injection experiment time because the target program shall be instrumented and compiled again for each fault injection target.

bility and correctness (i.e., data integrity) of application software. These emulated hardware-induced errors can be classified by using two metrics: fault location and type.

(i) Fault location. By using the location of the fault in the software-visible architecture state, hardware faults are classified into four types: (a) faults in ALUs, (b) faults in FPUs, (c) faults in registers (e.g., streaming multiprocessor registers in NVIDIA GPUs), and (d) faults in GPU hardware schedulers (e.g., streaming multiprocessor schedulers). These faults are emulated as errors in the architecture state (i.e., a program variable or control-flow decision). For example, a fault in an ALU is emulated by changing the value of a program variable that is computed by using at least one ALU operation. Here, we assume that the memory data transfers between GPU core and its cache/memory are reliably done. This assumption is valid in practice because both the cache/memory data and the data paths to on- and off-chip memory are protected by an ECC technique in current state-of-the-art GPUs.

(ii) Fault type. Fault can corrupt states of one or more transistors. An error in a single transistor can propagate and corrupt multiple bits in architectural states of a GPU. This injector can emulate both single- and multi-bit errors in architectural state (e.g., GPU register file). Although the latest GPUs support a SEC-DED ECC for register file, multi-bit errors can occur in register file and propagate to program states without being detected by the ECC technique. Supporting a stronger ECC has a hardware cost issue in practice. For example, while an SEC-DED ECC causes ~22% of extra space overhead when the protection unit is 32 bits (e.g., register), a DEC-TED ECC introduces ~41% of memory space overhead if the protection unit is the same.

5.9. Experimental Methodology

We use a GPU cluster where each node has an NVIDIA Tesla S1070 (4 GT200 GPU and 4GB memory per GPU) for the experiments. The Parboil benchmark suite v1 is used to evaluate application dependability.

In order to assess the performance overhead, we measure the time spent on GPU kernels, memory copies, and CPU-side codes. GPUs operate in synchronous mode when conducting this measurement. In practice, the measurement focuses on the GPU kernel execution time because the time spent on executing the CPU code and memory copy operations is similar regardless of used error detection technique. Note that even if the memory copy traffic is doubled, this does not increase the DMA time largely as long as the data size is

not excessively large as compared with the memory copy bandwidth between CPU and GPU memory (e.g., 4GB/s in PCI-E v2.0 with 8 lanes).

For the dependability evaluation, we emulate hardware faults (in various parts of hardware components) that propagate and corrupt single or multiple bits in an architectural state (register or memory). 20-50 virtual variables are selected in each benchmark program and faults are injected into each of the selected virtual variables. Fifty different error masks (randomly generated) are used for each variable in order to emulate single and multi-bit errors. In total, about 10,000 faults are injected into seven benchmark programs. Specifically, we perform 10,000 different fault injection experiments per application where each experiment runs a program and injects only one fault (either single- or multi-bit). Thus, in our experiments, error detection coverage p means that a fault in the used GPU programs can be either detected or masked with the probability of p if the characteristic of the fault is same as that of the used 10,000 faults.

The observed fault injection outcomes are classified into five types: (i) *failure*, a GPU kernel crash detected by the GPU runtime environment or a GPU kernel hang detected by the guardian process, (ii) *masked*, the output of a GPU kernel satisfies its correctness requirement regardless of the injected fault, (iii) *detected & masked*, the injected fault is masked but error detectors raise an SDC alarm, (iv) *detected*, the output of GPU kernel does not satisfy the correctness specifications and an alarm is raised by error detectors, and (v) *undetected*, if the output does not satisfy the correctness specifications but is not detected by error detectors.

5.10. Result

This section evaluates the performance and coverage of *HAUBERK* in comparison with (i) *Baseline*, without any custom error detection, (ii) *R-Naïve*, full duplication based on re-executing a GPU kernel twice, (iii) *R-Scatter*, an optimized full duplication exploiting data-level parallelism [DMZ09], (iv) *HAUBERK-NL*, *HAUBERK* only for non-loop codes, and (v) *HAUBERK-L*, *HAUBERK* only for loop codes.

5.10.1. Performance Overhead

Figure 5.14 shows the performance overhead of GPU kernels of seven HPC GPU programs (i.e., normalized to the baseline performance) when a same data set is used for training and testing. The average overhead of *HAUBERK* is 15.3%.

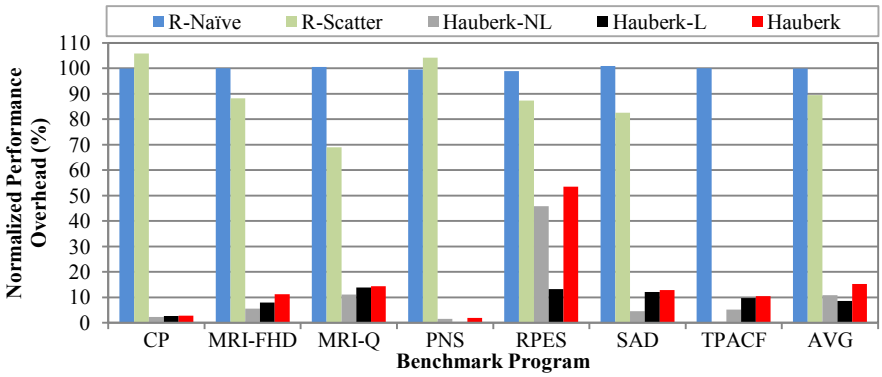


Figure 5.14. Performance overhead.

HAUBERK shows a significant performance overhead reduction as compared with *R-Naïve* and *R-Scatter*. The average overheads of *R-Naïve* and *R-Scatter* are 100% and 89%, respectively. This shows that the evaluation data reported in [DMZ09] holds even in more complex GPU programs. *R-Scatter* has a larger overhead in GPU than similar techniques for CPU program because its duplicated computation seeks same types of hardware resources or parallelism as the original computation, which is already heavily optimized in terms of used resources and parallelism. Note that statement duplication used in *R-Scatter* does not always double the performance overhead because the duplicated statements can be processed by using previously unused resources.

R-Naïve and *R-Scatter* have larger memory overheads than *HAUBERK*, which has only a small memory overhead (i.e., typically <10KB in both CPU and GPU memory spaces). *R-Naïve* doubles the CPU memory space to keep output data of the first and second executions of GPU kernel. *R-Scatter* doubles used GPU memory space and resources (e.g., global/shared memory and partly registers). This means *R-Scatter* is not directly applicable to programs that use more than half of one of these resources. For example, *TPACF* uses more than half of the GPU shared memory (e.g., 16KB total in the used GPU). Thus, we could not compile this program using the *R-Scatter* error detectors.

The average performance overhead of *HAUBERK* on the used benchmark is 15.3%. A large variation is observed in the performance overhead of *HAUBERK* on a particular program (*RPES*) where a large portion of GPU codes is sequential (i.e., non-loop). Excluding the performance overhead for *RPES*[31], the average overhead of *HAUBERK* is 8.9% where the minimum and maximum are 1.9% and 14.3%, respectively.

[31] We find that *RPES* is removed in the recent release of the Parboil benchmark suite because this type of program is not widely used as GPU program (i.e., inefficient due to the large portion of sequential code).

The performance overhead of *HAUBERK* is similar but not a straight sum of performance overheads of *HAUBERK-NL* and *HAUBERK-L*. This is because of common performance overheads (e.g., to deliver the control block between CPU and GPU and to manipulate the control block by placed error detectors).

The overhead of *HAUBERK-NL* depends on the portion of execution time spent on loops. For example, the overhead of *HAUBERK-NL* is exceptionally high in *RPES* because non-loop codes in this program form 75% of total execution time. In some benchmarks (i.e., *MRI-Q* and *MRI-FHD*), the overhead of *HAUBERK-NL* is larger than the contribution of non-loop codes to the total execution time because the duplication increases the register pressure (i.e., consequently increasing memory spill operations) and can interfere with the memory coalescing patterns in original program.

The overhead of *HAUBERK-L* has a relatively small variation because the same number (i.e., $Max_{var} = 1$) of variables is protected in each loop. The smallest overhead is observed when the protected variable is in integer type (i.e., *PNS*) thanks to the fast integer arithmetic speed. Note also that the overhead of *HAUBERK-L* is relatively small if the program (i.e., *CP*) has a self-accumulating variable (i.e., FP variable) in its loops. The fact that *CP* has larger overhead than *PNS* implies that value-range checker for FP data placed outside of a loop is an expensive operation in terms of performance overhead because FP variable has up to three value ranges to check.

5.10.2. Error Detection Coverage

Figure 5.15 shows the error detection coverage of *HAUBERK* for the benchmark programs and number of error bits when the same input data set is used for training and test runs. On average, 13.2% of injected faults can escape *HAUBERK* error detectors and lead to SDC errors. In other words the average detection coverage is 86.8%. If a system experiences two faults during its execution, the coverage of *HAUBERK* would be $(1 - (1 - 0.868)^2) = \sim 98.3\%$, assuming the two faults are independent. In the case of single-bit errors, on average, 35.6% of the errors are masked, 11.0% lead to a GPU program failure, and 21.4% are detected by the introduced error detectors. Out of the remaining 32%, 22.2% are detected but did not violate the application correctness, and 9.8% lead to SDC errors by bypassing the embedded error detectors.

The *detected & masked* error type is an error that changes an intermediate program state but does not make a large corruption in the output significant enough so as to violate the correctness requirements. Both the *detected* and *detected & masked* errors need a re-execution of the GPU kernel to diagnose false alarm because the golden output is not available in practice. This re-execution has relatively small impact on the average execu-

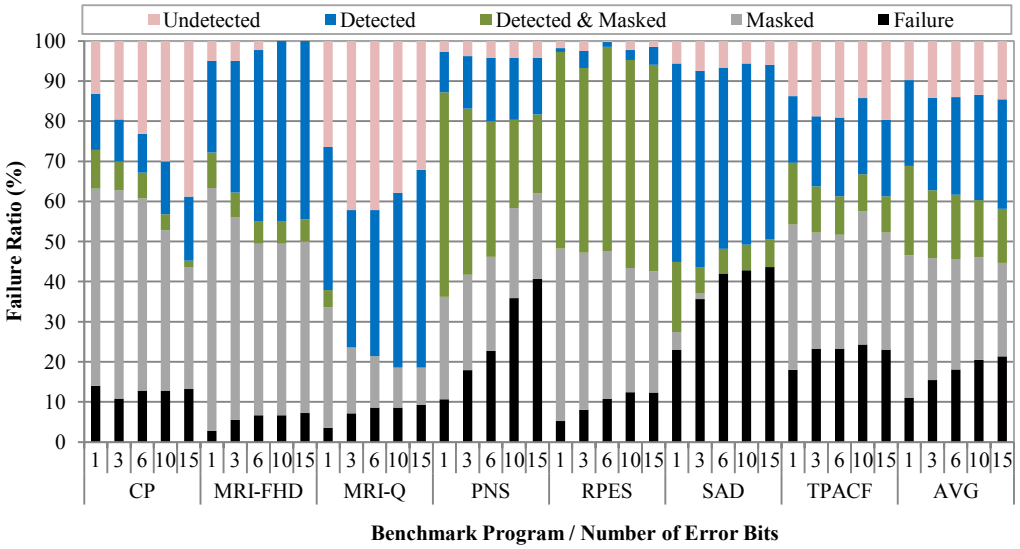


Figure 5.15. Error detection coverage of *HAUBERK-NL and HAUBERK-L*.

tion time of the GPU kernel because this fault can only happen if the GPU faces a hardware fault. In practice, the hardware fault rate is often low enough and the error recovery time is not long enough to impact performance.

The ratio of *detected & masked* type directly depends on the degree of strictness of output correctness requirement of application. For example, this ratio is low in *SAD* (i.e., an integer program) because it does not allow value errors in the output. This ratio is relatively high in *PNS* and *RPES*, where correctness requirements are relatively loose: $Max\{0.01,\ 1\%|GR_i|\}$ and $2\%|GR_i|+10^{-9}$, respectively. Here, $|GR_i|$ is i-th element of the golden output. Note that *MRI-Q* has stricter requirement than these two: $Max\{10^{-4}\text{Max}\{|GR|\},\ 0.2\%|GR_i|\}$ where |GR| means all elements in the golden output. [32]

Multi-bit errors typically increase the percentage of program failures and decrease the percentage of masked errors (see Figure 5.16). This is because when many bits are corrupted, this is likely to make a large value change in both FP and integer data as far as the number of corrupted bits is less than half of available bits. In Figure 5.16, regardless of an original value range, if the number of corrupted bits increases, the portion for >1E+15 (i.e.,

[32] Note that *MRI-FHD* and *MRI-Q* have similar algorithms and the same output correct requirements however these two programs show significantly different error sensitivities. Our analysis reveals that this is due to the difference in their input data where *MRI-FHD* in the used benchmark suite has a synthetic input data (all values are filled by zero) while *MRI-Q* has real input data. The following experiment shows the practical error sensitivity of *MRI-FHD* because it uses multiple real input data sets (see Figure 5.17).

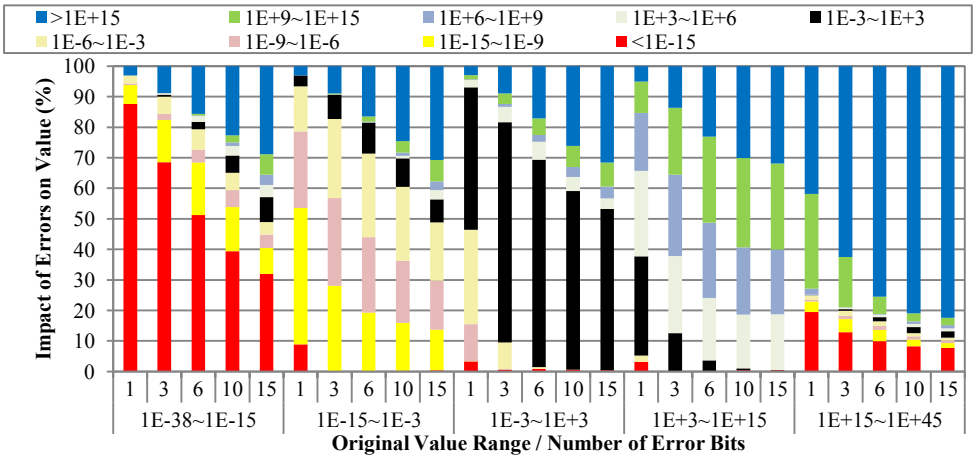


Figure 5.16. Changes in the magnitude of values after experiencing a fault depending on the orginal value range (of FP data) and error bit count.

value errors more than 10^{15}) gradually increases. This data is obtained by injecting faults into 33 million randomly-generated FP value samples. The same characteristic is observed in integer values.

Note that multi-bit errors do not always bring higher error detection coverage than single-bit errors when *HAUBERK*-generated error detectors are used. Some programs (e.g., *CP*) have lower coverage when many more bits are flipped. This is because multi-bit errors generally have higher non-benign error ratio (i.e., smaller masked error ratio), while many of these non-benign errors in these programs evade the provided loop error detectors. For example, if a corrupted variable is used as a divisor operand that computes another variable protected by a loop detector, a multi-bit error in the divisor operand variable can eventually reduce the protected variable value, i.e., less likely to be detected by the loop detector.

We find multiple cases where the GPU kernel hangs or faces a long execution time delay (i.e., a part of 'Failure' type in Figure 5.15). These failures are undetected by either *R-Naïve* or *R-Scatter*. An example case is when a loop iterator is corrupted (e.g., to a large negative number and the loop terminates if the iterator is bigger than a positive number) and the corrupted iterator does not cause a crash. Another example is specific to the *TPACF* implementation that uses a loop and performs a memory write operation until the write is successfully done and not overwritten by another thread (i.e., checked by reading the data back). If the address of memory write is corrupted to specific address ranges, the loop does not terminate because the corrupted address never returns the write requested value. Failures in these two cases are detected by the guardian process in *HAUBERK*.

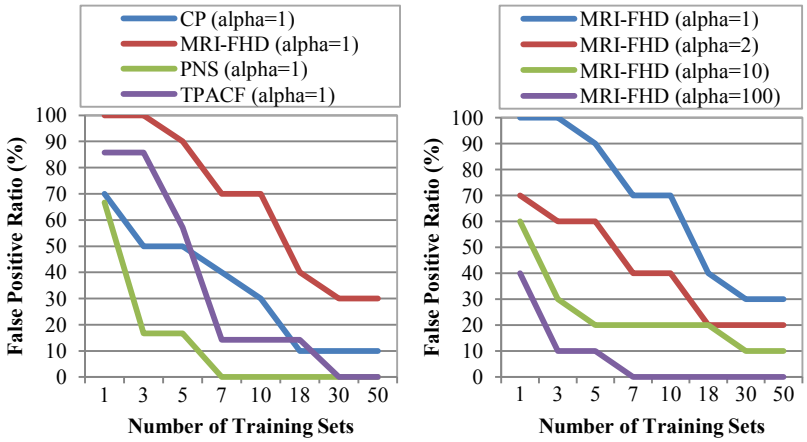


Figure 5.17. False positive ratio vs. Training count.

5.10.3. False Positive

We evaluate the false positives of *HAUBERK* loop error detectors by using different training and test data sets. Four benchmark programs are selected for this evaluation based on the availability of multiple data sets and their representativeness with respect to other programs. Out of 52 datasets prepared for each program, 50 are randomly selected and used for training and the remaining two are used for evaluating the derived detectors. This process is repeated 10 times to calculate average false positive ratio (see Figure 5.17).

We find that the false positive ratio largely varies depending on the program. For example, the measured false positive ratio of *PNS* becomes close to zero after executing seven training sets but that of *MRI-FHD* remains 30% even after running 50 training sets. This is because in the case of *PNS*, the program input represents parameters of a fixed simulation model and thus accurate detectors can be relatively easy to derive. In the case of *MRI-FHD*, the inputs are vectors and the output computation involves multiplication of the different vectors; thus, range-based detectors are not that precise.

In order to address detector imprecision, we investigate dynamic recalibration of the bounds (i.e., *min* and *max* values defining the bounds) used in the range detectors. The approach (described in Section 5.6 and 5.7) multiplies the bounds by *alpha*, a multiplication factor derived based on the monitored current false positive ratio.

(a) If alpha = 1. Even when *alpha* is 1, the false positive ratio quickly converges to a ratio less than 10% in the three out of the four evaluated benchmark programs (see Figure 5.17(left)). These three programs do not need to use the *alpha* larger than 1 (i.e., used 50

training input sets are sufficient). The detection ratio reported in Figure 5.17 corresponds to the *alpha* equal to 1.

(b) If alpha > 1. In the case of *MRI-FHD*, the false positive ratio does not quickly converge to below 10% if *alpha* is 1 (Figure 5.17(left)). Using a larger alpha value is needed even after processing more than 50 training input sets. Figure 5.17(right) shows the false positive ratio of *MRI-FHD* where the four curves are derived for *alpha* values of 1, 2, 10, and 100. When a large multiplication factor (i.e., *alpha*) is used, the false positive ratio decreases quickly after a small number of training sets. For example, for the *MRI-FHD* application applying *alpha* = 100, the false positive ratio becomes zero after executing 7 training sets. This shows that the adaptive technique to control value ranges used in the detector can efficiently manage the false positive ratios and consequently reduce the performance overhead.

We also analyze the impact of the selected *alpha* value on the detection coverage. The error detection coverage of *MRI-FHD* is 95%, 95%, 82.8%, and 81.6%, when the *alpha* is 1, 1000, 10000, and 100000, respectively. The value of *alpha* only affects the detection coverage of *HAUBERK* loop error detector. None or a small reduction (<0.5% decrease) in the error detection coverage is observed for *MRI-FHD* application when applying a multiplication factor (*alpha*) smaller than 1,000. This implies that the use of a large multiplication factor in the early stage of testing or training does not largely harm the error detection coverage because a fault in an FP or integer value often alters the data by orders of magnitude (e.g., $>10^6$ times, see Figure 5.16). A large increase (12.2%) in the undetected SDC ratio is observed when the *alpha* is set to 10,000. This threshold *alpha* value is not fixed but depends on multiple factors, including the iteration count of protected loop and the application output correctness requirement.

5.10.4. Instrumentation Time

We evaluate the instrumentation time of *HAUBERK* error detectors. On average, adding the *HAUBERK* instrumentation takes 81 seconds where the minimum and the maximum are 36 and 112 seconds, respectively, with the Parboil suite. The used machine has two 2.4GHz CPUs and 2GB DRAM and executes a Linux OS. This instrumentation time includes the C preprocessing time, parsing time, analysis time, and transformation time but excludes the time spent on C code beautifier.

The exact time spent on processing the *HAUBERK* transformers (i.e., placing error detectors in the intermediate representation) is 0.7 second, on average. The sizes of total program source code and GPU kernel code of each of the used benchmark programs are 579 and 266 lines, respectively, on average, before C preprocessing. If a source-to-source

translator is already used for other purposes (e.g., performance), the *HAUBERK* transformers only add a short delay (e.g., <0.7s, on average, per GPU kernel) to the total compilation time. The *HAUBERK* instrumentation can make small impact even if a target program is big and contains many GPU kernels.

The *HAUBERK* instrumentation is needed only after performance optimization and before testing. When developing an HPC program, most of the development time is spent on optimizing the program performance. After this optimization, developer typically runs an integrated stress testing. Because the *HAUBERK* instrumentation is for runtime fault tolerance, this instrumentation is added just before this final testing.

5.11. Summary

This chapter analyzed reliability problems in GPGPU platforms, focusing particularly on the design of efficient low-cost detection and recovery mechanisms for handling SDC (silent data corruption) errors. In order to tolerate SDC errors, customized error detection techniques are strategically placed in the source code of target GPU program so as to minimize performance impact and error propagation, and maximize recoverability. The presented *HAUBERK* technique is evaluated using a mutation-based fault injection tool (developed as part of this study) for automated reliability testing of commodity GPU devices. *HAUBERK* offers a high error detection coverage (~87%) with a small performance overhead (~15%).

Chapter 6.
Pluggable Watchdog: Transparent Failure Detection for MPI Programs

This chapter presents a framework and its techniques that can detect various types of runtime errors and failures in MPI programs. The presented framework offloads its detection techniques to an external device (e.g., extension card). By developing intelligence on the normal behavioral and semantic execution patterns of monitored parallel threads, these external error detectors can accurately and quickly detect errors and failures. This offloaded architecture allows us to use powerful detectors without directly using the computing power of the monitored system. The separation of hardware of the monitored and monitoring systems offers an extra advantage in terms of system reliability. Our experiment shows that the presented techniques, on average, cover 95.2% and 98.6% of the injected hardware faults with 1.8% and 8.4% performance overheads, respectively. The average failure detection latency is 117 milliseconds for crashes, 2.3 seconds for hangs, and 44 seconds for silent data corruption failures.[33]

6.1. Motivation

The growing scale and complexity of state-of-the-art parallel computers are having a negative impact on hardware and software fault rates. Early, accurate detection of resulting errors and failures is critical for the scalability of these computers. An undetected error can cause a hang failure of application software or a corruption in application output data. If such a hang or silent data corruption (SDC) failure is not detected shortly after its occurrence (e.g., within a checkpointing interval), the result is a checkpoint corruption problem. SDC failure is by definition undetected until the application termination, and consequently needs an application restart, delaying the expected execution time of application.

There is an engineering tradeoff between detection accuracy and overheads. In order to

[33]This chapter is accepted for publication: K. S. Yim, Z. Kalbarczyk, and R. K. Iyer, "Pluggable Watchdog: Transparent Failure Detection for MPI Programs," in *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*, 2013.

detect an error or failure, an extra computation must be performed. Such extra computation is typically introduced by creating redundancy in the time and/or space dimension. A larger amount of redundancy results not only in higher detection coverage but also in higher performance overhead and/or higher hardware cost. Strategically placed [FGA+10] and customized [YPS+11] error detection techniques show high efficiencies (e.g., coverage over overhead) in a statistical sense, but are still part of the tradeoff.

In this chapter, we explore the possibility of escaping that engineering tradeoff in a way that is transparent to application programs. The presented techniques make nonintrusive changes in target system hardware and software, making them applicable to parallel computers that use COTS (commercial-off-the-shelf) components. In the presented framework, the detection operations are offloaded to an external device (e.g., an extension card) of each compute node. When a Message Passing Interface (MPI) application is first launched, the *program flow extractor* derives a deterministic finite automaton (DFA) from the implementation of the application. The derived DFA captures all the legitimate execution sequences of selected types of instructions (MPI API call instructions) in the application. At runtime, a set of *signature monitors* run on a monitored computer and report the execution events of the selected instructions (in an MPI API call granularity) and other vital system events to a local monitoring device. A *monitoring device* (i.e., a PCI card in our prototype) uses its own embedded processor and memory and checks whether the application execution follows the program flow graph and complies with the past behavioral and semantic execution patterns. Thus, our framework can run powerful detection techniques without directly using the computing power of the monitored computer, other than to collect and report a set of events.

We design and evaluate a set of heuristics and value clustering techniques that can detect the behavioral and semantic errors and failures of MPI application processes. The termination state checker detects user process crash failures by forcing all MPI processes to terminate at one of the final states of their derived DFAs. Transition time checkers quickly detect user process hang failures by learning and enforcing the normal transition time intervals, where the transition time is time taken to transition from one DFA state to another state. These transition time checkers recognize the application execution phase, and use a tight timeout in each phase to reduce the hang detection latency. Output value checkers detect SDC failures by learning and enforcing normal value ranges of each type of application communication output messages. We present a dynamic profiling technique to make this output value checking practical in a new application running new input data.

We have prototyped our system on a parallel computer system by using an FPGA-based PCI extension card as a monitoring device. We conducted a fault injection experi-

ment to evaluate the presented techniques using eight MPI-based parallel programs. The techniques cover ~98.5% of faults, on average, with average failure detection latencies of 117 milliseconds, 2.3 seconds, and 44 seconds for crash, hang, and SDC failures, respectively. That means that only ~1.5% of random processor register faults, on average, evade the presented techniques and lead to failures. The average performance overhead is 1.8% for techniques that detect crash and hang failures and 6.6% for techniques that detect SDC failures. Moreover, the physical split of the monitoring device from the monitored computer has an extra advantage in terms of system reliability. Malfunction of a monitored computer does not directly affect execution of detection operations on the monitoring device, and vice versa.

The main contributions of this study are as follows:

- We summarize the experimental evidence that supports our design decision to detect failures in the highest layer of system abstractions (application software), especially if it is important to optimize the detection overhead (Section 6.2).
- We present a pluggable detection framework for modern high-performance computers without any intrusive changes to target system hardware and software (Section 6.3).
- We present a set of failure detection techniques that use a statically derived program flow graph and adaptively learning the normal behavioral and semantic execution patterns of a target program (Section 6.4).
- We develop a prototype system using only COTS components (Section 6.5).
- We evaluate the fault detection coverage and latency of the presented system via a fault injection study (Section 6.6.1) and measure the performance overhead of the presented system (Section 6.6.2).

This study demonstrates that by developing intelligence on the normal behavioral and semantic execution patterns of parallel application programs at a coarse granularity, external checkers can accurately and quickly detect non-benign faults with low runtime overheads.

6.2. Detection Target

We focus on detecting failures in the highest layer of system abstraction: the application process. In this chapter, we assume that the target application is a batch-mode parallel program launched from a command line shell of the header node and executes on top of parallel compute nodes.

We classify application failures into three types:

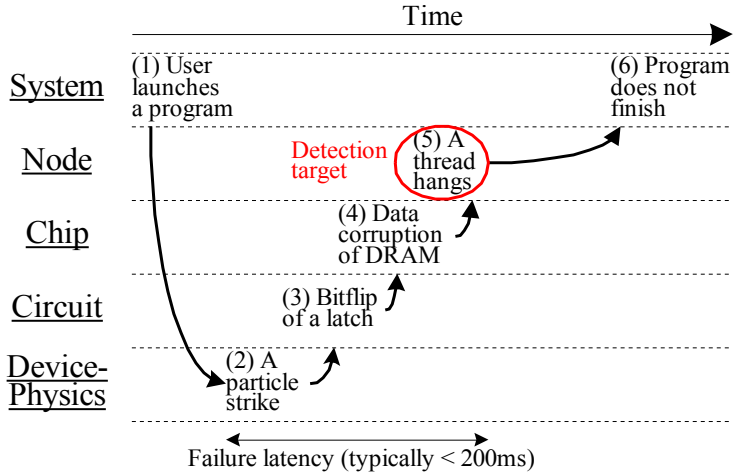


Figure 6.1. Detection target in fault-error-failure chains.

(i) Crash. Crash is an unexpected early termination of an application process. Crash comes with no or incorrect output data. Crash is by definition due to a detection of an error by a baseline error detection technique in the hardware, OS, middleware, or application.

(ii) Hang. Hang is defined as a continuous execution of an application process beyond its maximum, known normal execution time. Hang comes with no or incorrect output data. Hang is typically due to an undetected error that changes the control flow of a program and makes the program fall into an infinite loop. Note that a hang of an application process is due to a hang of itself or a hang of its underlying system software (e.g., OS or hypervisor) that is responsible for scheduling the application process. Thus, a user process hang detection technique can also detect crash and hang failures of both the OS and hypervisor.

(iii) SDC. SDC occurs when an application program terminates normally but produces corrupted final output data. We compare the output data of a golden (fault-free) run and a test run. If any values in the compared output data differ more than what is allowed by the application programmer or user (such correctness requirement is for example set to tolerate arithmetic errors of floating-point operations), we consider the test run to have incurred an SDC failure. SDC is typically due to an undetected error that may also have caused a data omission error or a timing error during the application process execution.

Those three types of application software failures are our detection targets. Our choice of detecting failures in the application layer is supported by the following evidence:

(i) Benign fault. In this chapter, we define a *non-benign fault* as a fault that successfully propagates to an application software state and leads to a user-visible failure. In each layer

of a system abstraction, a significant fraction of faults are masked and do not propagate to the upper layer of the system abstraction. For example, a transient fault in a combinational circuit of a processor is masked if the fault does not change the output data of the circuit at the exact same time the output is being saved in the following sequential circuit (see Figure 6.1). Similarly, a fault in a memory word is masked if the memory word has a dead variable. Previous fault injection experiments have extensively documented that masking effect in the device-physics, circuit, architecture, and software layers [KIR+99].

We believe that it is not always necessary to detect all faults in a low layer of system abstraction. It may be required only in mission-critical systems. Let us assume a technique that detects all low-layer faults (e.g., memory faults in the circuit layer), and restarts a program from a checkpoint if a fault is detected. Many of the detected faults are not visible to users, meaning that this technique is conservative in the sense that it wastes a lot of time on unnecessary restart and rework operations. Such waste is especially undesired in commodity computers, whose designs are tightly constrained by energy and cost efficiency requirements. Moreover, computers with a technique designed to detect and tolerate almost all faults in the low layers (e.g., dual or triple modular redundancy) still need a software failure detection technique if the computer nodes form a parallel or distributed system, because of their single points of failure.

(ii) Correlated faults. In commodity computers, many faults are spatially and/or temporally correlated, in part because of the use of COTS components, which are prone to permanent and intermittent faults. Memory errors observed in cloud datacenters are strongly correlated within the same memory module, and most of the uncorrectable errors are preceded by one or more correctable memory errors [SG06].

That correlation characteristic of faults implies that an imperfect detection technique can be effective in practice. For example, a technique with 90% fault detection coverage detects at least one non-benign fault with 99% probability if there are two non-benign faults, independent of each other.

(iii) Fail fast. Non-benign faults have the following two desirable characteristics that make it relatively easy for application-layer techniques to detect and tolerate them:

(a) Failure latency. Many previous fault injection experiments have found that most non-benign hardware faults have a short failure latency, where *failure latency* is the time from the introduction of a fault to the first occurrence of resulting failures in the same computer node. In x86 and PPC machines, >95% of non-benign transient faults in the processor and memory have failure latencies shorter than ~200 ms [GKI04]. Even shorter failure latencies are found with permanent faults (e.g., 99% of processor faults have failure latencies shorter than ~20 μs) [LRS+08]. In general, most non-benign hardware faults lead

to node failures in ~200 ms, while the exact percent of such faults varies depending on the configurations. The rest small portion of the undetected faults cause long latency failures and are analyzed in [YKI09].

The common failure latencies are several orders of magnitude shorter than common checkpointing intervals (e.g., >10 minutes). That means that the majority of failures can be tolerated through use of a checkpoint-restart technique, as long as there is a quick and accurate detection technique for these failures. Note that failures of a compute node are not always detected by the header node in parallel or distributed systems. In our preliminary study, ~14% of MPI user process crashes are undetected by the MPI header node (e.g., MPI middleware which is responsible for reporting the failure was in an unstable state when a type of crash signals arrives from OS), and an MPI process hang is detected by the header node after ~100 hours when an MPI middleware (MPICH2 v1.4.1) is used alone without any extra failure detection technique.

(b) Fault/failure location. The locations of faults and failures are highly correlated. For example, [GKI04] shows that 90–92.5% of failures are detected in the same OS module in which the faults were injected. The reason is that only a relatively small number of instructions can be executed within the short failure latency. When a failure is detected, a micro-rebooting of the affected software module can tolerate the failure if the induced errors did not propagate to outside the affected module.

In this chapter, we focus on errors and failures due to transient hardware faults. Considering the similarity in the types of possible errors and failures due to software faults, we believe that the presented techniques are generally applicable to protection against software faults.

6.3. Framework

The presented pluggable watchdog framework uses three types of new components in order to detect software failures.

(i) Program Flow Extractor (PFE). PFE is a post link-time static analyzer. PFE derives an execution behavioral model from a target program implementation (a binary file). The derived model is encoded as a DFA, where a transition represents the execution of a specific type of instructions and a state represents a program execution phase. The process must start from the start state and end at one of the final states of its DFA.

(ii) Signature Monitor (SM). SM is a software component that executes on each of the monitored computers. Multiple SMs exist. An SM measures a set of specific types of

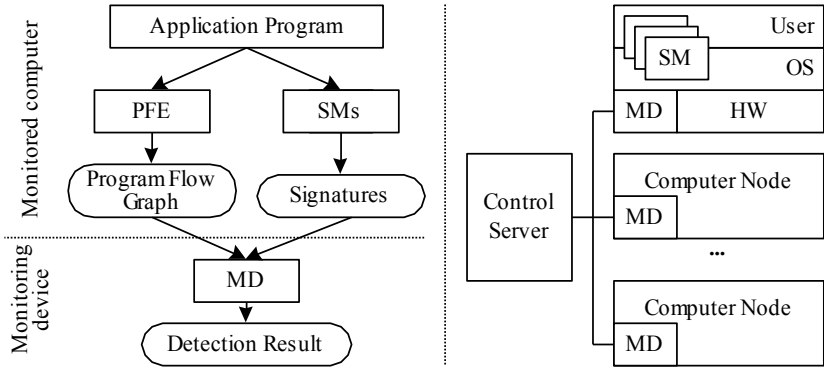


Figure 6.2. Failure detection model (left) and target system architecture (right) of the presented framework.

events in its host computer and reports them to the local monitoring device of its host computer. The provided SMs are implemented as OS modules or dynamically linkable user libraries. Thus, the provided SMs are easy to install.

(iii) Monitoring Device (MD). An MD is an extra hardware device in each of the monitored computers (e.g., an extension card or a separate chip in the motherboard). A set of error detection techniques execute on the MD of each node. The detection techniques use the dynamically measured execution signatures and/or statically derived program flow graph of the target application (see Figure 6.2, left).

The presented framework has a control server node that controls the MDs of all compute nodes of a target parallel computer (see Figure 6.2, right). The control server runs a GUI-based control program, which uses TCP and UDP communications to, for example, relay program flow graph files and failure detection results between the header node and MDs. If there are many compute nodes, one may use multiple control servers and organize them in a tree-like structure.

The presented framework has a pluggable, nonintrusive architecture. In other words, all three components are easily installable on commodity computers. PFE is a user-level program, and SMs are an MPI middleware wrapper library and a set of OS kernel modules. If the target system uses a dynamically linkable MPI library, the user can install the provided SM library by replacing the existing library. The user must also install the provided PFE program in the header node, and an administrator must install the provided SM kernel modules in all compute nodes.

6.3.1. Program Flow Extractor (PFE)

The presented PFE is designed to derive a program flow graph that captures all the legiti-

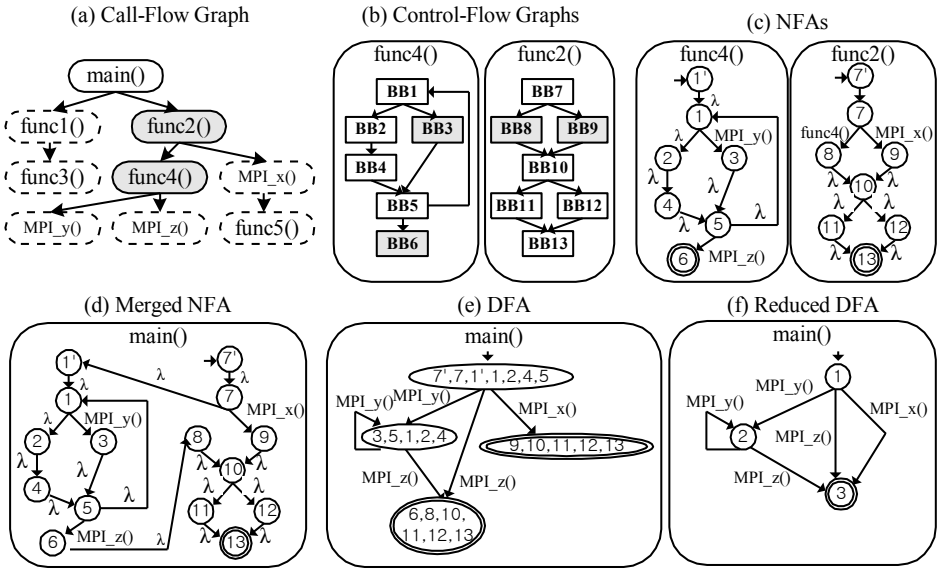


Figure 6.3. Program flow graph derivation process.

mate execution sequences of all the targeted types of instructions in a given program. The user specifies the target instruction types. Let us assume that the target program is an MPI program, and that the target instruction types are call instructions whose callee symbol name starts with "MPI_". These types are to identify all call instructions to MPI API functions. The program flow graph derivation process has three major steps:

(i) Preprocessing step. PFE reads the binary file of the target application program. PFE builds the complete call-flow graph (i.e., flows between functions) of the target program and complete control-flow graph (i.e., flows between basic blocks) of each function in the program. To do so, PFE disassembles the binary file and builds an intermediate representation (IR) data structure of the disassembled code. The data structure of IR encodes the program call- and control-flows and is easily re-organizable.

(a) Call graph. PFE traverses all functions that are reachable from the main function, and constructs the call graph of these reachable functions. Regardless of the programming language used (e.g., C/C++, Fortran), the used linker generates the binary code by using main as the starting function of a program. All unreachable functions are eliminated from the IR data structure.

PFE eliminates functions whose execution does not have any direct or indirect dependency with the execution of any of the target instructions. PFE traverses all functions in the call graph and marks all functions that contain at least one target instruction. Figure 6.3(a) shows an example; gray boxes are marked functions and dotted boxes are unreachable

functions. PFE then recursively marks all the direct and indirect callers of the marked functions. Unmarked functions are eliminated from the call graph IR data structure. If the MPI API call instructions are specified as targets, all MPI API functions themselves are also eliminated if they do not internally call any other MPI API functions.

(b) Control-flow graphs. For each function in the reduced call graph, PFE builds a control-flow graph. In a control-flow graph, a vertex is a basic block, and an edge is a control-flow instruction between two basic blocks (see Figure 6.3(b)). Each vertex contains a bit that indicates the inclusion of at least one target instruction (see the gray boxes in Figure 6.3(b)).

Let us assume that the control-flow graph of function *foo* is notated as CFG(*foo*) = (*V*, *E*, *s*) where *V* is a set of basic blocks in *foo*, *E* is a set of control flows (e.g., $v_i \rightarrow v_j$) between the basic blocks in *V*, and *s* is the entry basic block of *foo*.

(ii) NFA derivation step. From the control-flow graph of each of the left functions, PFE derives a nondeterministic finite automaton (NFA).

(a) Basic block splitting. For each control-flow graph, PFE splits basic blocks for which the total count of the contained target instructions and function call instructions is more than one. For example, if a basic block has a target instruction *A*, call instruction *B*, and target instruction *C* in sequence, this basic block is split into three basic blocks, where each of the split basic blocks contains just one of instructions *A*, *B*, or *C*. The split basic blocks are connected so as to follow the original execution sequence. In this example, PFE marks only the first and third split basic blocks to indicate the inclusion of a target instruction.

(b) NFA conversion. PFE converts each of the split control-flow graphs to an NFA. Table 6.1 shows the algorithm that converts a control flow graph CFG(*V*, *E*, *s*) to an NFA($\underline{V}$, $\underline{E}$, $\underline{s}$, $\underline{f}$) where $\underline{V}$ is the set of states, $\underline{E}$ is the set of transitions, $\underline{s}$ is the initial state, and $\underline{f}$ is the final state of the NFA. There is a one-to-one mapping between the NFA states and the exits of the CFG basic blocks except for the starting state of the NFA (see Figure 6.3(c)), making it produce a compact NFA.

The algorithm starts by pushing a pair of the starting state *s* of CFG and the initial state $\underline{s}$ to a set *Q* (line 3). *H* is a hash table that uses a state in CFG as the key and returns the pair state of the key in NFA. In the first iteration, *H* is empty. That creates a new NFA state $\underline{v'}$ (that is paired with *v*) and inserts $\underline{v'}$ to *H* as the hash value of *v* (lines 10–13). If *v* has a callee function, this callee is set as the callee of *v'* (line 11) that notates a point to merge two NFAs. The algorithm then checks whether *v* contains a target instruction (line 14). If the condition is true, the label of a new transition is set from $\underline{v}$ to $\underline{v'}$ as the event name of the contained target instruction (lines 15 and 18). If it is false, the transition label

Table 6.1. CFG to NFA Conversion Algorithm.

Convert a CFG (V, E, s) to an NFA (V, E, s, f)
01: $\underline{V} := \emptyset$, $\underline{E} := \emptyset$, $\underline{F} := \emptyset$, $Q := \emptyset$, $H := \emptyset$
02: $\underline{s}$:= create the starting state in NFA
03: enqueue $\underline{s}$ to $\underline{V}$, enqueue (s, $\underline{s}$) to Q
04: **while** Q $\neq \emptyset$
05: | (v, $\underline{v}$) := dequeue Q
06: | **if** v is hashed in H **then**
07: | | $\underline{v'}$:= lookup the hash value of v in H
08: | | *visited* = true
09: | **else**
10: | | $\underline{v'}$:= create a new state in NFA
11: | | set callee of v as callee of $\underline{v'}$ if any exists
12: | | insert $\underline{v'}$ as the hash value of v to H
13: | | *visited* = false
14: | **if** v is marked **then**
15: | | l := get the event name of the target instruction in v
16: | **else**
17: | | $l := \lambda$
18: | $\underline{e}$:= create a transition from $\underline{v}$ to $\underline{v'}$ and label as l
19: | enqueue $\underline{e}$ to $\underline{E}$
20: | **if** *visited* = false **then**
21: | | E' := a set of transitions in E that exits v
22: | | **if** $E' = \emptyset$ **then**
23: | | | enqueue $\underline{v'}$ to $\underline{F}$
24: | | **else**
25: | | | **for** each $e \in E'$
26: | | | | v' := the control-flow target basic block of v
27: | | | | enqueue (v', $\underline{v'}$) to Q
28: $\underline{f}$:= create a new state in NFA
29: **while** $\underline{F} \neq \emptyset$
30: | dequeue v from F
31: | create a λ-transition from v to $\underline{f}$
32: **return** ($\underline{V}$, $\underline{E}$, $\underline{s}$, $\underline{f}$)

is set to λ, indicating that this transition can be made without consuming any event (line 17). If v has no control-flow target basic block, v is an exit basic block, and thus the algorithm inserts $\underline{v'}$ to $\underline{F}$ (lines 22–23). Otherwise, the algorithm inserts the control-flow target basic blocks of v (up to 2 basic blocks) to Q with the pair NFA state of v (lines 25–27).

If Q is implemented as a queue, the algorithm searches basic blocks of CFG in the breadth-first manner (lines 4–5). If Q is implemented as a stack, the algorithm uses the depth-first search. If the first v' is the same as v (e.g., a loop that has only one basic block) and the breadth-first search is used, in the next iteration, the condition at line 6 is met, and thus a transition is made from an existing NFA state ($\underline{v'}$ in the previous iteration) to itself using the same label. Because this is not the first visit to v (line 20), the algorithm skips the rest of the operations.

The algorithm finally creates a new state (f) as the final state of NFA (line 28). There

Table 6.2. Algorithm to Merge NFAs.

```
Generate a merged NFA
01: H1 := create an empty hash table, H2 := create an empty hash table
02: (V, E, s, f) := generate NFA of the main function
03: return Merge-NFAs(V, E, s, f, H1, H2)
Merge-NFAs(V, E, s, f, H1, H2)
04: (V'', E'', s'', f'') := clone of (V, E, s, f)
05: for each state v in V
06: |  for each transition e in E that exits v
07: |  |  v' := target state of e
08: |  |  if v' has a callee function then
09: |  |  |  if H1.lookup(symbol name of the callee of v') < N then
10: |  |  |  |  H1' := clone of H1, H2' := clone of H2
11: |  |  |  |  (V', E', s', f') := generate NFA of the callee function of v'
12: |  |  |  |  if H1'.lookup(symbol name of the callee of v') is empty then
13: |  |  |  |  |  H1'.insert(symbol name of the callee of v', 0)
14: |  |  |  |  |  H2'.insert(symbol name of the callee of v', {s', f'})
15: |  |  |  |  H1'.update(symbol name of the callee of v', H1'.lookup(v')+1)
16: |  |  |  |  create a new transition from f' to v'
17: |  |  |  |  change the transition target state of e to s' from v'
18: |  |  |  |  (V''', E''', s''', f''') := Merge-NFAs(V', E', s', f', H1', H2')
19: |  |  |  |  insert (V''', E''', Ø, Ø)  to (V'', E'', s'', f'')
20: |  |  |  else
21: |  |  |  |  {s', f'} = H2.lookup(symbol name of the callee of v')
22: |  |  |  |  create a new transition from f' to v'
23: |  |  |  |  change the transition target state of e to s' from v'
24: return (V'', E'', s'', f'')
```

must be at least one final state of NFA in F (line 29). For each state in F, the algorithm creates a transition to a new state f (lines 30–31). Optionally, one may extend the algorithm to check the size of F and directly use f in F as the final state of NFA if $|F| = 1$.

(iii) DFA conversion step. PFE finally merges the derived NFAs and converts the merged NFA to a reduced DFA. PFE prints the reduced DFA as a flow graph of a target program.

(a) Merging NFAs. Table 6.2 is the algorithm to merge all the converted NFAs to a new NFA, where *H1* is a hash table that tracks the call count of each function, and *H2* is a hash table that keeps the first-produced NFA instance of each function. This algorithm starts from the NFA of the main function (lines 2–3). For each transition *e* in the current NFA (lines 5–6), if the transition target state has a callee function and the callee exists in the reduced call graph (lines 7–8), it creates a λ-transition from the final state of the NFA of the callee to the transition target state (lines 11, 16), and changes the transition target state of *e* to the initial state of the callee NFA (line 17). If this callee is seen less than N times, it replicates the callee NFA by a recursive function call (lines 9, 18). This recursive function call returns a merged sub-NFA. The algorithm removes the initial and final state marks from the returned NFA before merging it to the current NFA instance (lines 4, 19).

Lines 20–23 are to reuse the pre-generated NFA states if the same callee is seen ≥*N* times. Figure 6.3(d) shows an example of merged NFAs.

This merge algorithm has a key difference from the existing technique [GSV05], which was developed for intrusion detections. We replicate the NFA of a callee that has a cycle in the call graph (e.g., recursions) for a certain number of times (*N*). This reduces impossible paths in the derived program flow graph that is useful when a target program implementation uses a recursion (e.g., for depth first searching or back tracking). Also, since we monitor the MPI API calls, we can convert the program flow graph into a deterministic form, unlike system calls [WD01], whose call sequences depend on runtime factors (e.g., context switching). Note that this does not mean we always produce a deterministic program flow graph, because of asynchronous events (e.g., signal), *setjmp*, function pointers, and dynamic libraries, which must be manually handled. Note that to the best of our knowledge, there is no previous work that exploits program flow graph (similar to our approach) for hardware-induced error detection.

(b) DFA conversion. PFE converts the merged NFA to an equivalent DFA and optionally reduces the converted DFA by using well-known algorithms from automata theory (see Figure 6.3(e) and 6.3(f)). PFE encodes the reduced DFA by an extended adjacency list notation that lists all the possible transition event (MPI API call types) and destination state pairs of each source state. The encoded DFA marks initial and final states.

PFE saves the encoded DFA to a regular file named in the form p.dfa if the original binary file name is p. That extraction process is invisible to the MPI program developers and users. When a user launches an MPI program, the first process executing on the MPI header node checks whether there is p.dfa file in the current directory. The provided MPI wrapper library (see Section 6.3.2) performs this checking and the following operations on behalf of the first process when it calls the *MPI_Init* function. If `p.dfa` does not exist, the first process runs the PFE program with the binary file `p` and the pre-specified target instruction types to generate `p.dfa`. The first process then sends the content of `p.dfa` to the control server (see Section 6.4.1(i)) before continuing its execution.

PFE extracts a program flow graph from the implementation of a target program (but not a design). The reason is that our target is a hardware fault that occurs at runtime (not at design time). As long as the programming model is SPMD (Single Program Multiple Data), PFE can statically derive program flow graph and enforce it on each parallel execution instance. In adaptive MPI where parallel execution instances can migrate from one node to another, migration has to be tracked, and MDs of the source and destination compute nodes must communicate the tracking information of the migrated instance.

We have implemented PFE from scratch in C++. Our PFE implementation has 4,970

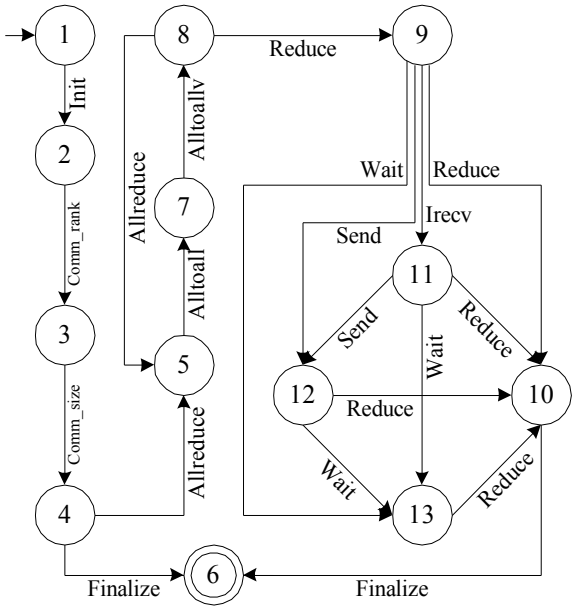


Figure 6.4. A program flow graph of IS (integer sort) in NPB.

lines of codes. We used PFE to extract the program flow graph from each program in NAS Parallel Benchmark (NPB) v2.3. The total number of states is 5 to 14 per program. The total number of transitions is 6 to 50 per program. Figure 6.4 shows a DFA of the integer sort program in NPB derived by PFE.

6.3.2. Signature Monitor (SM)

A set of SM agents run on each of the monitored compute nodes. There are two types of SM agents: system and user (see Figure 6.5). Each SM agent measures and reports the signatures of software to its local MD by using the SM runtime in its host OS kernel.

(i) SM agent. The provided system and user of SM agents are implemented as a kernel module and user-level library wrapper, respectively. Both of them can be installed without any intrusive modification in monitored system.

The system agent monitors selected system events and sends them to its local MD via the SM runtime module. The reported event data contain information about the event type, logical processor ID, user process ID, and occurrence time. Some system agents preprocess event data before reporting. For example, a system agent computes interrupt event counts per each logical processor for a time interval and only reports the counter values to the local MD. Such an agent can use not only numerical sum but also average, variation, median, histogram, and other data aggregation methods.

There is one *default system agent* that monitors the call events of the selected kernel functions by rewriting the kernel code in the memory. When this agent is first executed, it disassembles the kernel codes in the OS kernel image file and identifies all the call instructions to each of the selected kernel functions. It then replaces the call target address of each of the identified call instructions with the address of the corresponding wrapper function defined inside this agent.

The default system agent has one wrapper function for each type of the selected kernel functions. The selected kernel functions are *panic()* and *do_die()* to detect kernel crash events, and *do_exit()* to detect user process termination events in Linux. The wrapper of *panic()* and *do_die()* sends a kernel crash event to the monitoring device by using the urgent delivery mode, where the urgent delivery is a non-blocking operation that immediately sends the event data by using statically pre-allocated resources. The wrapper function of *do_exit()* identifies the index of the current process (e.g., MPI communication rank) if the current process is one of the monitoring target processes, and saves the identified index in the memory (see Section 6.4.1(ii) on how target processes are identified). The SM runtime module sends the saved *do_exit* event to the local MD at the next reporting time together with other user-level signature events, so that all buffered events are delivered in the order in which they occurred.

The user agent for MPI programs is an MPI library wrapper that monitors all MPI API function call events at runtime. At the entry and exit of each MPI API function, this user agent composes an event data packet (containing the event type, current process ID, and other parameters of interests) and sends the composed packet to the MD. If the MPI library is provided as a dynamically linkable library (DLL), this rewritten MPI wrapper library is installed on a target system through replacement of the existing DLL file. If the MPI library is provided as a statically linked library, the binary file of a target program is regenerated through relinking of the object files of a target program with this rewritten MPI library without any program source code modification.

(ii) SM runtime. The SM runtime is an interface between SM agents and the local MD. It provides SM agents with common mechanisms, and consists of two components:

The *SM runtime kernel module* is an OS kernel module that provides interface functions to SM system agents. Among its interface functions, *register_event_handler(x,y)* registers a callback function *x* to a specified type of events *y*. The supported system events include system call entry/exit, interrupt handler entry/exit, and context switching. *report_event(x, y)* sends data *x* to the local MD. If the urgent delivery flag *y* is set, this function immediately sends *x*. If it is not, this function saves *x* in the memory to send the saved data to the local MD at the next reporting period. The SM runtime kernel module uses a

dedicated kernel thread for the periodic reporting. Thus, the caller of this function does not need to wait until the completion of a send operation. When small-size data are frequently sent, this kernel module can still fully utilize the communication bandwidth to its local MD, because the data are concatenated in the memory.

The *SM runtime library* is a user library that allows SM user agents to use the SM runtime kernel module without any extra memory copy and system call operations. The runtime library maps the exposed buffer memory of the runtime kernel module to the virtual address space of each of the host user processes of this library (using the *mmap* system call in Linux). The library and the runtime kernel module provide the same event reporting interfaces to SM user agents and SM system agents, respectively. By calling those provided interface functions, each user agent directly accesses the memory buffer of the runtime kernel module without any context-switching operation. All accesses to the memory buffer are synchronized through use of a custom synchronization primitive (test-and-set instruction) because the buffer is shared by all the system and user agents.

6.3.3. Monitoring Device (MD)

An MD is hardware external to a monitored computer. Errors in a monitored system can thus rarely propagate to its local MD, which could be, e.g., a board-level chip, a chipset extension, an extension card, a network fabric, or another computer node. In our prototype, we use a PCI extension card that has: (a) a PCI target controller for communication with its monitored system, an embedded processor (SPARC v8), and DRAM chips to execute the detection techniques, (b) a network interface chip (e.g., Ethernet) for communication with upper-layer MDs (e.g., control server), (c) local/remote storage (e.g., flash memory or network file system), and (d) a private power cable (see Figure 6.5).

The PCI target controller exposes a set of registers (other than the standard PCI control registers) to its monitored system. The SM runtime module uses the registers to initiate copy operations to its local MD and issue an interrupt to the MD when the copying is done. The embedded processor runs an embedded OS (Snapgear Linux v2.6), handles interrupt requests of its monitored system, and uses the following software components to process the data.

(i) Event queue. The event queue is a kernel module of the embedded OS of the MD that receives data from the monitored system. The event queue classifies data received from the monitored system and forwards the classified data to a proper event-processing engine.

(ii) Event-processing engine. Event-processing engines are classified into two types: system and user. System engines process system-level events (e.g., sent by the default sys-

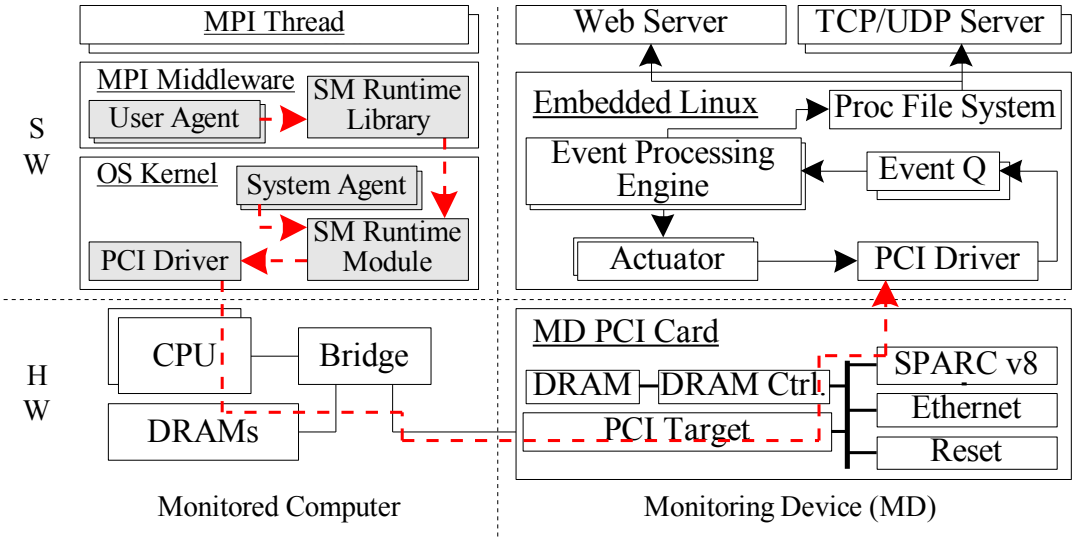


Figure 6.5. Software architecture of signature monitors and hardware architecture of a monitoring device.

tem agent). For example, a system engine monitors OS crash events and reports them to the control server. Another system engine monitors all user process crash events and forwards them to a user engine for failure detection. User engines process user-level events. A user engine receives the program flow graph of a target program from the control server when the program is launched, and traverses the DFA instances of the program flow graph as the program executes and generates event signature data (see Section 6.4.1(i) for the DFA dissemination process). We use multiple user engines to support the detection techniques presented in Section 6.4.

(iii) Actuator. Actuators take actions on the monitored system to avoid, detect, or recover from failures. Actuators are classified into three types. One type controls the physical hardware of the monitored system (e.g., performs soft reboot to tolerate a transient fault). Another type sends commands to the system or user agents in order to execute them on the monitored system (e.g., make a checkpoint or migrate a user process between CPUs). The last type logs the processing results or reports the results to upper-layer MDs.

6.4. Techniques

Our detection techniques are classified into three types:

6.4.1. Crash Detection

We detect user process crash failures by checking the *termination state (TS)* of each monitored process.

(i) DFA dissemination. If an MPI program is launched, its first process sends the DFA file (`p.dfa`) to the control server and waits for an acknowledgment. The control server sends an acknowledgment after successfully disseminating `p.dfa` to all of its registered MDs. Our wrapper library forces all MPI processes to execute the *MPI_Barrier* function before actually executing the *MPI_Init* code. After the first process has completely disseminated p.dfa and called *MPI_Barrier*, all MPI processes, including the first one, exit *MPI_Barrier* and continue their executions. As an alternative, we also provide a script that performs this DFA derivation operation before executing the mpirun command.

(ii) Monitored process registration. Note that when a target program starts and calls *MPI_Init()*, our MPI wrapper library also extracts the process identifier and MPI communication rank of its host user process and send that information to the default system agent. The default system agent maintains a list of monitored processes of its host compute node, and reports the list to its local MD so that when one of the registered processes terminates, the local MD can recognize that it is a monitored process.

(iii) Termination State (TS) checker. As the program runs and generates user signatures (e.g., call events of MPI APIs) in each MD, a user event-processing engine traverses the DFAs of all monitored processes of its monitored computer. If the process termination event is forwarded to this user engine, the engine checks the following to detect crashes.

The checked condition is whether each monitored process terminates at one of the final states of its DFA. For example, in the DFA shown in Figure 6.4, "1" is the start state, and "6" is the final state. For the termination event of process *P* (*do_exit* event), if the current state of the DFA of *P* is a non-final state (not "6"), our TS checker declares the crash failure of *P* and reports this to the control server.

Let us assume that there is a control-flow error which changes the condition of a branch. If such control-flow error also changes the sequence of MPI API call events of the program and the changed sequence does not have a valid path in the derived program flow graph, such control-flow error is always detected by the TS check because the TS checking naturally includes the *transition destination* (TD) checking. However, it is possible that some control-flow errors do not change the condition of any branch instruction. Such error is either benign or leads to a hang or an SDC failure. The following two subsections describe hang and SDC failure detection techniques.

6.4.2. Hang Detection

Our *transition time (TT) checker* detects hang and timing failures of user processes by using the normal patterns in interoccurrence times of various granularities of user events.

(i) Interoccurrence time. Our TT checker measures the interoccurrence times between any two consecutive user events. It measures them at one of the following granularities and keeps the max interoccurrence time per event pair group.

(1) Hang-State-Node. In this case, the interoccurrence time is the transition time of each ordered pair of DFA states. One max transition time is kept for each source DFA state. The kept time of a state *S* is the maximum observed transition time from *S* to any of its legitimate destination states.

(2) Hang-State-Thread. In this case, the interoccurrence time is the same transition time as in *Hang-State-Node*, but one maximum transition time is kept per state and per thread.

Figure 6.6 shows the measured transition time distributions of eight out of sixteen processes of the IS program running with its largest input dataset, where each graph is for a specific source DFA state. State "6" has no transition time data because it is the final state. We also do not use the transition time of the initial state "1" for error checking. One can define the transition time of the initial state as a time from a parallel process is created to the *MPI_Init* function is called by that process. In that case, the transition time of the parallel process can have a relatively large variation in practice.

(a) There is a relatively large variation between different source states. For example, source state "4" always has a transition time between 1.5 and 2.1 seconds, while source state "7" always has a transition time of less than 3 ms. That shows that the source state is an effective classifier for learning about and enforcing the normal transition times. For example, if we are using the short normal transition time pattern of state "7", when no transition is made for ~20 ms, our TT checker declares a hang failure if the current state of a process is "7", but does not declare a hang failure if the current state is "4".

(b) The transition times of the same source state are similar. In Figure 6.6, states "5" and "8" have the lowest peaks, but still, their transition times are numerically similar. In case of the state "5", all the transition time samples are less than 310 ms. The similarity between threads is partly because of the SPMD nature of our benchmark programs, and the shortened communication intervals that come as a result of strong scaling. Based on this observation, we configured our TT checker to use the *Hang-State-Node* granularity in our experiment.

(ii) Timer setup. When a new transition event occurs, our TT checker computes the timeout value as $\max\{M \cdot P_{margin}, P_{mindl}\}$, where M is the maximum transition time of the current state, P_{margin} is the timeout margin parameter, and P_{mindl} is the minimum detection

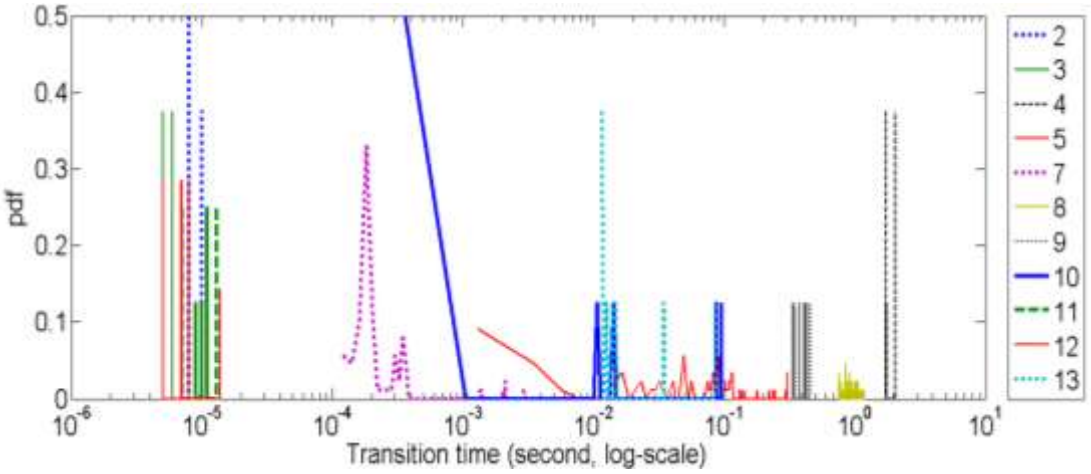


Figure 6.6. Transition time distribution of IS in NPB; the legend label indicates the exiting state; y-axis is the probability density function (pdf).

latency parameter that includes the maximum reporting period of the SM runtime. Our TT checker sets a timer for the current state of each of the current processes by using the computed value. If the current process has a pending timer, our TT checker updates the timeout value of the pending timer.

(iii) Timeout. If a timer expires, it indicates that there was no transition in a monitored process within its maximum expected transition time interval, and our TT checker adds 1 to the node hang process count. If the node hang process count is above a certain threshold (e.g., 25% of monitored processes in the node), our TT checker declares a user hang failure and reports this to the control server. It is our empirical observation that if an MPI process hangs, all the other processes of the same program are also likely to experience hang failures in a short time interval (e.g., deadlock). Thus, by using the hang process count, our TT checker can manage the false positive ratio without harming the detection ratio.

(iv) Adaptive management of timer. When a monitored user process terminates (*do_exit* event), our TT checker resets any previously set timer of the corresponding process. In practice, the detection latency and false positive ratio of our TT checker depend heavily on the P_{margin} and P_{mindl} values. Our TT checker thus adaptively changes these values as it gains more confidence in M. In our prototype, the initial value of P_{mindl} is 30 seconds (considering the dispatch time of parallel processes), and P_{margin} is 10. For each M, as it measures K more transition time samples, it reduces P_{mindl} by R percent down to 50 ms, and reduces P_{margin} by R percent down to 2. Here, the low bound of P_{mindl} (50 ms) shall be double the length of the reporting period of the SM runtime. The minimum values of P_{mindl} and P_{margin} set the minimum bound of the detection latency of our TT checker. K and R are

the learning efficiency parameters. Larger values of K and smaller values of R make our TT checker more conservative that results in a smaller false positive ratio and a longer detection latency for the early phase of monitoring. We set K to 1 and R to 1% in our experiment. Users set the initial parameter values for each program execution and send them to MDs (e.g., as in DFA dissemination). At runtime, the user event-processing engine of each MD, which implements our TT checker, maintains one parameter value set per DFA state if the *Hang-State-Node* granularity is used.

6.4.3. SDC Detection

Our *output value (OV) checker* learns normal patterns in the communication output data values to detect SDC failures.

(i) Output events. The MPI program uses certain types of MPI APIs to send messages to other MPI processes (e.g., *MPI_Send*, *MPI_Reduce*, *MPI_Alltoall*). Such an output API function sends an array of the same type of data, where the data type is specified as another parameter of the API call. This API convention make our SM user agents (implemented as MPI wrapper library) possible to derive output data types at runtime without rewriting the application program.

(ii) Value histogram. A user agent computes a histogram by using all the data array values of each output API call and reports the histogram to the local MD. The user agent computes the logarithm of each data value of the array (v) and converts the logarithm value to an index of a histogram bucket by using (6.1) where $\mathrm{MM}(v) = \max\{\min\{v, \propto\}, \beta\}$.

$$I(v) = \begin{cases} \mathrm{MM}(\lfloor log_{base}|v|\rfloor)+\propto, & v < 0 \\ \mathrm{MM}(\lfloor log_{base}|v|\rfloor) + 2 \propto +\beta + 1, & v \geq 0 \end{cases} \quad (6.1)$$

Here, each histogram bucket keeps values with the same order of magnitude. *base* is 2 or 10 in our experiment, depending on the data type. The $\propto$ and β parameter values also depend on the data type. If the data type is IEEE 754 single-precision floating point, the value range of each sample of data in v is approximately $\pm[2^{-126}, 2^{127}]$, and the value range of $\log_2|v|$ is [-126, 127]. Based on that, $\propto$ can be set to less than 126 and β to less than 127. Considering the low probabilities of having extremely small or large values, we set $\propto$ to 63 and β to 64 in our experiment, because the value range of $I(v)$ then becomes $[0, 2(\propto+\beta) + 1 = 255]$ (i.e., a compact and small memory space for encoding).

(iii) Profiling. We obtain $\mathcal{H}=\{H_1, H_2, \ldots, H_n\}$ by profiling a target program to use $\mathcal{H}$ for SDC detection. H_t is the histogram of the t-th type of output data, and is an array of $2(\propto+\beta)+1$ tuples where $\propto$ and β are parameters for the type t data values. Each tuple uses 1 bit in static profiling and 2 bits in dynamic profiling. Each tuple of H_t indicates whether at least one value is seen for the corresponding histogram bin during profiling ("1") or not

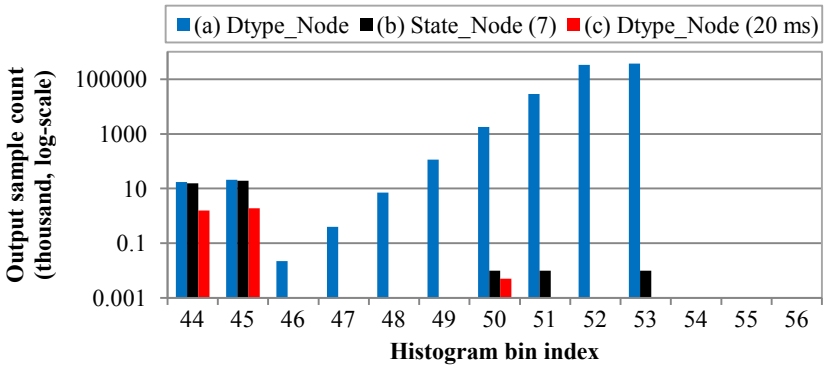


Figure 6.7. Profiled histogram examples of IS.

("0").

Our OV checker declares an SDC failure if $I(v) = i$ but $H_t(i)$ = "0" where v is any output value seen during testing. The reason is that there is at least one new value that is at least an order of magnitude different from all the profiled values of the same program.

We support four different profiling granularities of $\mathcal{H}$.

(1) SDC-Edge-Node. $H_t(i)$ indicates whether there is at least one thread that can produce a value for the i-th histogram bin of H_t. Here, the type t is uniquely chosen through use of the data type and corresponding DFA edge of the MPI output operation.

(2) SDC-Edge-Thread. This is similar to *SDC-Edge-Node* except that this keeps a different $\mathcal{H}$ for each process.

(3) SDC-Dtype-Node and *SDC-Dtype-Thread.* These are similar to *SDC-Edge-Node* and *SDC-Edge-Thread*, respectively. The difference is that the type is chosen using only the data type of the MPI output operation.

(a) Static profiling. Static profiling obtains $\mathcal{H}$ by running the same target program with input data that are the same or similar to that used in testing or with many common input data. IS that runs with the largest input dataset ("C" in NPB v2.3) produces 7 different types of histogram bins, where 6 are for unsigned short (bin count of 16 in our experiments) and 1 is for single-precision floating-point data (bin count of 256). When *SDC-Dtype-Node* is used, $\mathcal{H}$ has two histograms (see Figure 6.7(a) for a part of the histogram of unsigned short data). If *SDC-Edge-Node* is used, $\mathcal{H}$ has seven histograms (see Figure 6.7(b) for a histogram part of unsigned short data from DFA state "7"). There is a large difference in bins between 46 and 53 because there are other DFA states generating output data of the same unsigned short type. Table 6.3(a) shows a part of the histogram data structure for Figure 6.7(a).

(b) Dynamic profiling. We present a dynamic profiling technique that at runtime learns

Table 6.3. A Part of Histogram of IS from Index 44 to 56.

Bin#	44	45	46	47	48	49	50	51	52	53	54	55	56
(a)	1	1	1	1	1	1	1	1	1	1	0	0	0
(c)	1	1	0	0	0	0	0	0	0	0	0	0	0
(c3)	1	1	2	2	2	2	1	2	2	2	0	0	0

* (a): SDC-Dtype-Node, (c): SDC-Dtype-Node (20 ms), (c3): (c)+Distance 3

the normal output value ranges of a target program ($\mathcal{H}$). When an output MPI API function is called, a user agent first identifies the type of the output event (t). As in static profiling, there are two different identification granularities. *SDC-Edge* has a histogram array for each pair of a DFA edge and an output data type. *SDC-Dtype* keeps an array for each output data type regardless of the DFA edge.

At runtime, a user agent computes the histogram of the output data that are being sent. If the histogram is for the t that is first seen, H_t is initialized as a "0"-filled array. For each value (v_x) in the given output data array, the agent computes $i_x=I(v_x)$ and sets $H_t(i_x)$ to "1". One H_t is maintained either per thread or per node. Thus, dynamic profiling also supports four different histogram tracking granularities similar to those of static profiling.

Our dynamic profiling technique uses two parameters: distance and dynamic profiling time. If distance is d, this technique also marks d numbers of left and right neighbor bins of any histogram bin that is set to "1". Here, marking means that if and only if such a neighbor bin is set to "0", this bin is set to "2". Figure 6.7(c) shows a part of the dynamically profiled histogram of unsigned short type output produced by an edge exiting state "7" in IS when 20 ms has elapsed after the output of this type of data was first seen. Because of the short dynamic profiling time, it only captures the values in bins indexed as 44, 45, and 50. Table 6.3(c) shows the part of the derived H_t of the histogram bins shown in Figure 6.7(c). The row "(c3)" shows H_t when the distance is 3 for the histogram in Figure 6.7(c), where "2" indicates the neighbor bins. The histogram shown in Table 6.3, row "(c3)" has neither false positives nor false negatives, but could have false negatives if the distance is 4 due to the bin "54".

If a value for a neighbor bin is seen, this is not treated as an SDC failure, because it may be a result of insufficient dynamic profiling. When a new type of a histogram is first produced, our technique creates a timer and sets its timeout value to the given dynamic profiling time. Until the dynamic profiling timer expires, our technique does not declare an SDC failure, even if there is v such that $I(v) = i$ but $H_t(i)$ = "0". If the timer expires, our technique then starts to declare SDC failures, because profiling has been done for a sufficient time and the confidence in the obtained $\mathcal{H}$ has become high enough according to the given dynamic profiling time value. We optimize the dynamic profiling time parameters

of eight parallel programs in Section 6.6 and show that longer distances need shorter dynamic profiling times.

6.5. Experimental Methodology

We have developed a prototype system of the presented framework and techniques. We measured the performance overhead of the system using the prototype, and performed a set of fault injection experiments to evaluate the fault detection coverage and latency.

(i) Prototype. Our prototype of the presented system consists of two compute nodes, one header node, and one control node. Each compute node has two 6-core 2-way SMT processors (i.e., x86 ISA) with 32GB of DDR3 DRAM. Both 10Gbps SDR InfiniBand (RDMA) and 100 mega bps Ethernet interfaces are used to connect these two nodes. Each node executes a Linux kernel v2.6.39 and uses MPICH2 v1.4 as the MPI middleware, GCC v4.3.4 as the C/C++ compiler, and Intel Fortran compiler v12.1 as the Fortran compiler.

(ii) Experimental setup. We used NPB v2.3 as the benchmark suite, which included one C program and seven Fortran programs. Each compute node had 24 logical processors, and each pair of logical processors shared the processor resources due to the SMT architecture. In each node, we ran 8 MPI processes by statically binding them to specific logical processors in such a way that no MPI processes shared the same SMT logical processor.

(iii) Fault injection framework. We extended a software-implemented fault injection framework [YKI09] in order to emulate an error in a specific MPI process. As fault injection targets, we randomly selected a set of instructions to set breakpoints (about 700 to 2,200 instructions, depending on the program code size). We set a breakpoint either right after a program launch or after a certain time of execution (e.g., to analyze how the detection latency changed as a program ran for a longer period of time). We emulated a random single-bit error in a randomly selected processor register just before the selected instruction was first executed after the breakpoint was set. The studied registers were general-purpose (*eax*, *ebx*, *ecx*, and *edx* in x86 ISA) and special-purpose registers (*esi*, *eflags*, *ds*, *esp*, and *eip*). Only a part of the selected instructions were actually executed (i.e., faults were activated) during our experiments. We use only the activated fault samples (2,352 from 8 programs) in our evaluation.

For MPI processes, we add a software component to the breakpoint-based injector. The component selects a specific MPI process as a fault injection target. The added *target*

selector component is an MPI library wrapper that instruments two MPI functions: *MPI_Init()* called when each MPI process starts, and *MPI_Finalize()* called just before each MPI process terminates. The code added to *MPI_Init()* reads the target process selection information from a file and checks whether the identifier of current MPI process (or thread) is the same as the specified one. In the implementation, we use the MPI communication rank to specify an MPI process. If current thread is the target, this added code extracts the process identifier of the current process in OS. This code writes the extracted identifier and other fault injection command information to the fault injector module by using the proc file system interface. The code added to the *MPI_Finalize()* function checks the injection results by reading the injector status information via its proc file system interface (e.g., whether the fault was activated).

6.6. Results

We evaluate the detection coverage, detection latency, and performance overhead of the presented techniques. Coverage is defined as 100% minus the percentage of faults that evade the protection and lead to failures. Detection latency is the time from the injection of a fault to the first detection of a resulting failure by the control server.

6.6.1. Fault Detection Coverage and Latency

Figure 6.8 summarizes the fault detection coverage. "A" indicates that only the baseline fault detection techniques were used. On average, ~59.5% of faults were not manifested (i.e., benign faults). In the figure, the y-axis starts from 40%, as all faults below 40% were not manifested. Among the rest of the manifested faults, 30.6%, 5.1%, and 4.8% led to user crash, user hang, and SDC failures, respectively, on average.

When all the presented techniques were used in tandem (technique labeled "D"), the average fault detection coverage was 98.6%. When the TS and TT checkers were only used, the average detection coverage was 95.2%. Specifically, the TS checker detected all user crash failures, and the TT checker alone detected all user hang failures that were found and tested in our experiments. However, the presented techniques missed a non-negligible portion of SDC failures.

(i) Crash. The TS checker detected all the found user process crash failures (see Figure 6.8(B)). All the used benchmark programs outputted the execution result before calling *MPI_Finalize*. PFE marked all the transition targets of that *MPI_Finalize* event as final states. Thus, if a process crashes before calling an *MPI_Finalize*, the TS checker always detects such crash failures. It is possible for a process to successfully output the result data

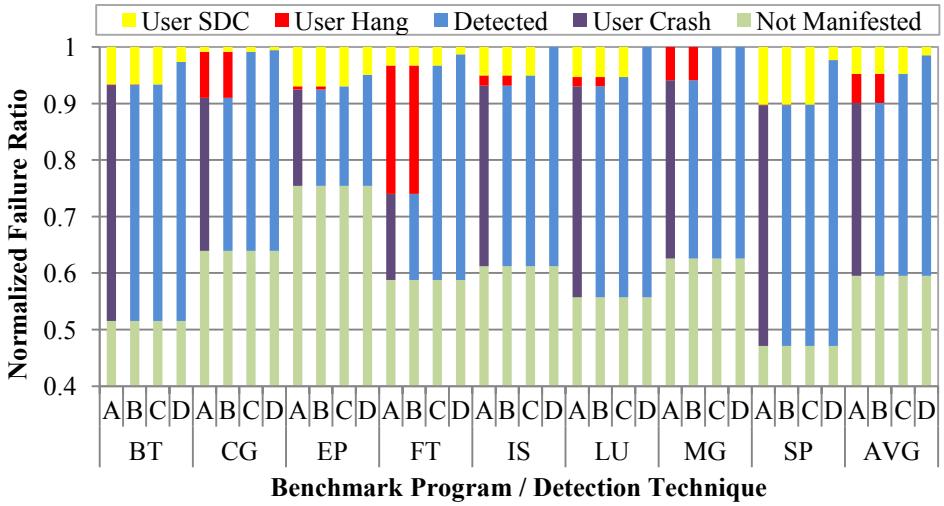


Figure 6.8. Fault detection coverage

* "A" in x-axis is for baseline, "B" is for TS, "C" is for TS+TT, and "D" is for TS+TT+OV.

but subsequently crash before completely calling an *MPI_Finalize*. The TS checker still detects such crash failures, although the detections are false positive. Because a fault must occur between a few instructions (e.g., from the last instruction of the output operations to an *MPI_Finalize* call instruction) in order to cause such crash failures, the TS checker has an extremely small false positive ratio, and we did not observe any cases in our experiments. If a process crashes after calling an *MPI_Finalize* but before completely terminating itself, the TS checker, by design, does not detect the failure. The reason is that we consider such crashes to be normal executions (or SDC failures), because the process has produced the result data and transitioned to a final state. It is possible that the output is corrupted, which is however is no longer a detection target of the TS checker but of the SDC detection techniques. Note that if an application program terminates early due to a detection of an error by its built-in error detector, the TS checker does not detect this event, because the execution still follows the design and implementation of a program. In such a case, the application outputs an error message and immediately returns the shell prompt to the user.

Figure 6.9 ("Crash") shows the cumulative probability distribution of the detection latencies of all crash failures. The average of the crash detection latency was 117 ms. The TS checker detected 95% and 99% of crashes before 56 ms and 710 ms, respectively. That detection latency includes the extra time to process and send events to the control server, which is a reason why the detection latency is slightly longer than for the baseline detec-

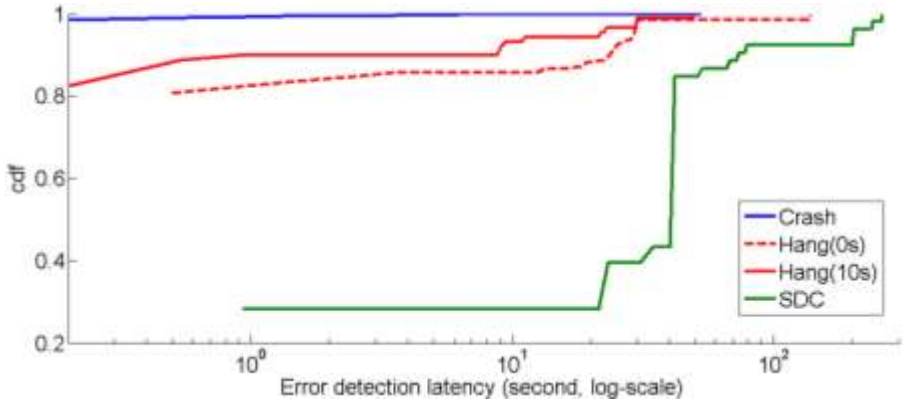


Figure 6.9. Cumulative error latency distribution.

tion techniques of OS [GKI04][YKI09].

(ii) Hang. One of our TT checkers (configured as *Hang-State-Node*) detected all hang failures (see Figure 6.8(C)) in our experiments. All the observed hang failures changed the control flow of at least one process in a way that made the process stop calling MPI API functions. In the most common case, first a process that had experienced a hang failure stopped calling MPI API functions, and then all the other processes of the same program stopped calling MPI API functions, because of their communication dependencies. Another common case was that processes in only one node crashed, while processes of the same program on the other compute nodes did not. In that case, all other processes were likely to stop, because of their communication dependencies on the crashed processes. However, the header node did not always recognize such crash failures of a compute node.

By checking the interoccurrence time between consecutive MPI API call events, the TT checker alone could detect all hang failures in our experiments. We found that checking the transition destination state did not increase the fault detection coverage. One reason is a relatively rare execution frequency of MPI API function calls (e.g., coarse-granularity monitoring), while in practice an instruction or a basic block is likely to form an infinite loop if there is a transient fault.

In this chapter, hang failure detection means that hang failure is eventually detected within our monitoring time. A technical challenge is to minimize the detection latency and false positive ratio. We selected the parameter values of the TT checker (given in Section 6.4.2) in such a way that the TT checker had no false positives with the used benchmark programs and still had a short detection latency. The TT checker gradually reduced its timeout interval as the monitoring time increased and it collected many more transition time samples.

We evaluate the hang detection latency as a function of the fault occurrence time (i.e., a

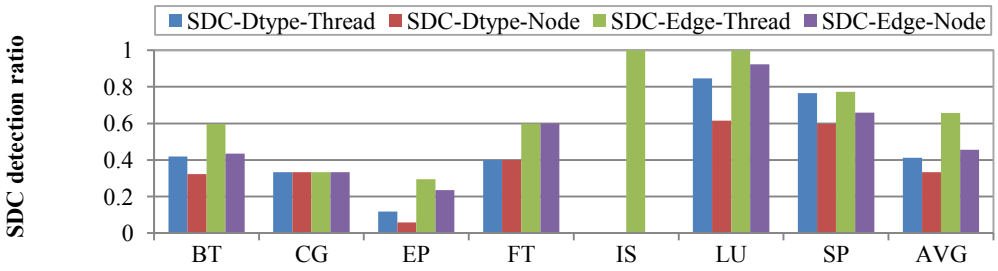


Figure 6.10. SDC detection ratio; x-axis is benchmark program.

wait time to set breakpoints). Figure 6.9 also shows the distributions of the hang detection latency, where "Hang (0s)" refers to the times when the breakpoints were set right after a program was launched, and "Hang (10s)" refers to the times when the breakpoints were set after a program had executed for 10 seconds. Note that some breakpoints of "Hang (0s) " were triggered after 10 seconds, as the breakpoint instruction is infrequently executed, and injecting the same type of faults (e.g., the same breakpoint, register type, and error bitmask) at different time points does not always lead to the same types of failure (e.g., some fault is not activated). In "Hang (0s)", the average, 95th percentile, and 99th percentile of the detection latencies were 5.4, 29.4, and 137.8 seconds, respectively, and in "Hang (10s)", they were 2.3, 21.9, and 42. This shows that the TT checker quickly learned the normal behaviors and set relatively tight timeout intervals for many DFA states.

The presented hang detection technique differs in the following ways from the previous techniques. In a hang detection technique that embeds heartbeat generation codes in the source code, the heartbeat generation code sends a heartbeat to another computer node or resets its local watchdog timer. This technique not only needs a source code modification but also loses detection coverage in many cases. For example, if the heartbeat generation code is embedded inside a loop, this technique cannot detect a hang failure that repeatedly executes the loop. Another technique uses the execution instruction count instead of the execution time. Instruction counts are useful if multiple user processes run in a time-shared manner. Note that supercomputers and high-performance distributed computer systems run only one program at a time on each computer node. Instruction counts must be explicitly measured and reported by an entity residing in the monitored computer, while the execution time is universally known fact (i.e., something an external monitoring device naturally knows).

(iii) SDC. The measured SDC failure ratios of BT, CG, EP, FT, IS, LU, MG, and SP were 6.6%, 0.8%, 7.0%, 3.2%, 5.0%, 5.3%, 0%, and 10.2%, respectively. The output verifier of MG found no SDC failures in our experiments because MG is an approximation

algorithm, and its output verifier allows relatively significant value errors.

(a) Static profiling. In static profiling, the average SDC detection ratios of *SDC-Dtype-Thread*, *SDC-Dtype-Node*, *SDC-Edge-Thread*, and *SDC-Edge-Node* were 41.2%, 33.3%, 65.7%, and 45.5%, respectively. Finer-granularity output data checking realized higher detection ratios. For example, *SDC-Edge-Node* had 12.2% higher coverage than *SDC-Dtype-Node*, clearly showing the benefits of tracking program execution phases and enforcing different error conditions. IS clearly exemplifies that. In IS, *SDC-Edge-Thread* detected all SDC failures, but both *SDC-Edge-Node* and *SDC-Dtype-Thread* detected none. For example, normal values of some processes (or states) are abnormal in some other processes (or states) that can only be distinguished by the finest-grained checking.

The ratios shown in Figure 6.10 reflect profiling done using the same (or highly similar) input data; the program is deterministic in terms of its output data values. There are many scenarios in which the same program runs many times with similar input data (e.g., weather forecasting) in parallel and distributed systems. If a target program runs with new unknown input data, one may profile the value histograms by using multiple known common input datasets and use the OR operator to merge the histograms of each dataset. In such a case, if the histogram profiling is done for an insufficient number or combination of input datasets, finer-grained checking can have higher false positive ratios. As the profiling is done for many more input datasets, both the detection coverage and false positive ratio go down, which has been well-studied for another value range checking technique [YPS+11].

(b) Dynamic profiling. The presented dynamic profiling is useful in providing detection for a new program that runs with a new input dataset. Figure 6.11(a1) and 6.11(a2) show the normalized dynamic profiling time as a function of the distance parameter value. Here, the normalized dynamic profiling time is the ratio of the shortest execution time needed to obtain the histograms equivalent to the statically profiled histograms, and the program normal execution time. For the same distance, *SDC-Dtype-Node* in general requires less dynamic profiling time than *SDC-Edge-Node* does, because *SDC-Dtype-Node* maintains a smaller number of histograms and thus uses many more samples per histogram than *SDC-Edge-Node* does. In *SDC-Dtype-Node*, if the distance was ≥30%, all the benchmark programs could obtain the golden histograms in negligibly short dynamic profiling times (e.g., <1 ms). When *SDC-Edge-Node* was used and the distance was 0.1%, three programs (CG, EP, IS) required <1.5% of the execution time for dynamic profiling. Three other programs (FT, MG, SP) required relatively short dynamic profiling time when the distance was 1% or 5%. The remaining two programs (LU, BT), however, required a long dynamic profiling time if the distance was less than 10%.

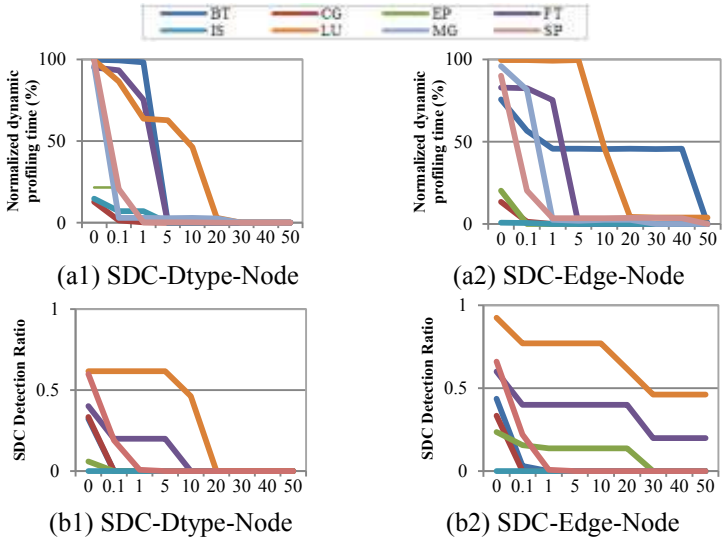


Figure 6.11. Distance vs. Profiling time ((a1) and (a2)) and Distance vs. SDC detection ratio ((b1) and (b2), no data for MG)

* x-axis is the distance relative to the total histogram bin count (%).

Figure 6.11(b1) and 6.11(b2) show the SDC detection ratio as a function of the distance parameter. Note that this gives the low bound of the SDC detection ratio for a given distance parameter value. In general, *SDC-Edge-Node* gives a higher detection ratio than *SDC-Dtype-Node* because of its finer-grained checking. In *SDC-Edge-Node*, when the distance was 0.1%, four programs (EP, FT, LU, SP) maintained relatively high detection ratios. Among the four, FT and LU still maintained the high detection ratios when the distance was increased to 20%. Considering the tradeoff between the distance and the dynamic profiling time, at least three programs (EP, FT, LU) can take advantage of this dynamic profiling.

Figure 6.9 also shows the cumulative detection latency distribution of SDC failures by the dynamic profiling technique with the distance parameter of 0.1%. SDC detection latency is much longer than the crash and hang detection latencies. The majority of SDC detection latencies were longer than 1 second in our experiments because of the time taken by errors to propagate to and show up as communication output data. The average SDC detection latency was 44 seconds. 95% and 99% of SDC failures were detected within 202.5 and 260.8 seconds. All the observed SDC failures were detected within 2,500 seconds, which is close to the total execution time of some programs. The reason is that some SDC errors were detected later, when the program exchanged the final output data at the end of its execution. This implies that in order to further reduce the SDC detection la-

Table 6.4. Performance Overhead.

	BT	CG	EP	FT	IS	LU	MG	SP	AVG
I*	0.1%	1.2%	0.9%	0.6%	6.6%	0.2%	5.1%	0.1%	1.8%
I+II*	0.6%	14.5%	1.0%	1.3%	32.6%	3.8%	10.9%	2.6%	8.4%

* I: Call Event Monitoring, II: Output Data Value Monitoring

tency, one can monitor the types of events that are executed more frequently than the communication outputs.

Failures of MDs do not impact their host application behavior because MDs are implemented as separate hardware devices. Failures of MDs can be detected using existing hierarchical heartbeats if control servers are organized in a tree-like structure by using the existing control networks for heartbeat reporting.

There is an engineering tradeoff between checking frequency and detection latency. The presented techniques are still part of that engineering tradeoff. However, long detection latency failures in the presented techniques are not difficult to recover because the presented techniques isolate the failures to each MPI process. Specifically, SDC failures are detected before they propagate to another MPI process. Although it may take a long time to detect, still the recovery technique can tolerate the detected SDC failure by locally restarting one MPI process for example without having to restart all other MPI processes.

6.6.2. Performance Overhead

Table 6.4 shows the performance overhead of the presented techniques, where "I" refers to the overhead of the crash and hang detection techniques, and "II" refers to that of the SDC detection techniques. The average performance overhead of "I" was 1.8%, which is much lower than that of "I+II", 8.4%, because the histogram calculation uses computation- and memory-intensive operations and produces a relatively large volume of data to report to the local MD.

The performance overheads were exceptionally high in CG, IS, and MG. For "I" alone, these three programs have higher overheads than the rest of the programs. This indicates that these three programs call MPI API functions more frequently. Among the frequently called functions, some were communication output APIs. The performance overhead of "II" was higher in IS than CG or MG because of the larger output data size (and/or higher execution frequency of output MPI APIs in general). We manually visualized the executions of IS and EP using the MPE and Jumpshot tools. IS frequently uses the *MPI_Allreduce* and *MPI_Alltoallv* operations during its execution in order to exchange different pieces of data among all pairs of parallel processes, and the time spent on *MPI_Alltoallv* forms >50% of the total execution time (showing the significance of out-

puts). On the other hand, EP uses no MPI API functions to perform computation (because EP is an embarrassingly parallel program) and uses *MPI_Allreduce* and *MPI_Barrier* functions only once to merge the computation results of all processes. This explains why EP had a low overhead of 1%.

There are many existing techniques that can significantly reduce the communication overhead between SMs and their local MD (e.g., DMA). These not only reduce the communication overhead but also allow us to offload a larger portion of detection operations (e.g., histogram) to MDs. Some techniques can reduce the impact of SM operations on the parallel program performance (e.g., jitter mitigation).

6.7. Related Work

At least two types of existing fault detection techniques are related to the presented framework and techniques (see the summary in Table 6.5).

(i) Program-level embedded checker is error detection code embedded in monitored software. The placement and customization of error detection code can be automatically done if compiler-based techniques are used. As summarized in [YPS+11], such techniques typically incur >15% performance overhead. Many of them rely on static profiling data. [YPS+11] showed ~7 well-chosen input datasets are sufficient to control the false positive and negative ratios in parallel programs. In this chapter, we explored a dynamic profiling technique that profiles the normal value patterns of a program at runtime. Our approach is based on the fact that the same piece of program code is likely to produce similar output values regardless of how long it has been executed.

ABFT [HA84] is a superior example of an embedded checker, because it offers high detection coverage and low runtime overhead in fundamental linear algebra operations. Recently, ABFT was extended for other matrix operations (e.g., sparse matrix multiplication [SKB12], dense linear system solver). ABFT however requires a significant amount of manual effort to rewrite application programs, while the presented framework automatically customizes the failure detectors. Not only that, program-based detectors share HW with monitored programs while the presented framework splits them (e.g., to isolate HW faults to each other).

(ii) External checker uses an observer that measures specific types of execution signatures of the target programs (e.g., OS event or hardware performance counters) and sends them to an external checker for analysis and monitoring. The observer can be embedded in the target program source code or can be an extension of runtime software (e.g., JVM ex-

Table 6.5. A review of existing hardware fault detection techniques.

Technique	Description	Comment
Full duplication	*Creates spatial and/or temporal replica of program (e.g., DMR, re-execution)*	*A necessary technique for high availability systems but is suitable for COTS due to hardware change or large energy overheads (Overhead >100%, Coverage ~100%, Intrusiveness high)*
Optimized duplication	*Optimizes the execution of the replica using parallelisms in HW (e.g., EDDI, SWIFT)*	*The most effective in VLIW and superscalars but less effective in fully-optimized parallel programs (Overhead 40-80%, Coverage ~99%, Intrusiveness medium for SW technique)*
Selective protection	*Selects a set of computer states or events that are likely to show the presence of errors (e.g., selective duplication, program likely invariants, processor symptom)*	*Detects many common failures efficiently but such states or events do not always exist (e.g., for data errors) and often sensitivity of program states depends on how they are used (Overhead 30-70%, Coverage 70-90%, Intrusiveness medium)*
Manual customized protection	*Manually customizes software algorithm or fault tolerance techniques for fault detection and/or recovery (e.g., ABFT)*	*Typically shows the highest efficiency if a program uses specific types of operations (e.g., linear algebra) and such manual development cost is acceptable (Overhead <5%, Coverage >95%, Intrusiveness high)*
External monitoring with artificial signatures	*Offloads detection to an external detector of program and uses artificially collected program execution events (e.g., watchdog, runtime verification, anomaly-based)*	*Detects many errors if the semantic gap to monitored software is well managed, and if the offloading is done to a separate HW, detection operation itself becomes reliable (Overhead <20%, Coverage 70-90%, Intrusiveness low)*

tension). An external checker can run on the same computer node as the monitored program or another computer node. Some uses statistical clustering to detect abnormal behavior of the target program [AMA+11]. Some other [CR07] uses the formal properties of a target program and verify the properties at runtime.

Our presented technique belongs to this type; specifically, it is most similar to watchdog processors. A watchdog processor checks the control flow of a program by snooping on the memory bus. Some previous work presented an extended watchdog processor to improve the coverage [MM88] and to support multitasking systems. Our presented system can be treated as an extension of watchdog processors for the commodity computers that use large-size on-chip caches and many-cores. Note that the control-flow checking of watchdog processors generally offers higher coverage than symptom-based techniques

(that can rarely detect SDCs because SDCs are due to corruptions in application data). We further present heuristics and value clustering techniques to capture the behavioral and semantic execution patterns of applications and detect SDCs. We also show that middleware API calls are effective signatures to detect software failures.

6.8. Summary

We have designed and implemented a software failure detection framework for COTS-based parallel computers. The framework splits the hardware of monitored and monitoring devices in order to isolate hardware faults to each other. This architectural design helps us to use powerful failure detectors without directly using the computing power of monitored devices. The techniques detect many non-benign hardware faults (98.5%) with a small performance overhead (8.4%) because the techniques recognize the execution phases of applications using the statically derived program flow graphs, and learn and use the normal behavioral and semantic execution patterns of each execution phase of applications. Our prototype implementation and evaluation results demonstrate feasibility of the presented framework and techniques and justify large-scale experiments.

Chapter 7.
FDX: Fault Tolerant, Programmable Voter (Architectural Support for Error Detection)

This chapter presents a fault-tolerant, programmable voter architecture for software-implemented N-tuple modular redundant (NMR) computer systems. Software NMR is a cost-efficient solution for high-performance, mission-critical computer systems because this can be built on top of COTS devices. Due to the large volume and randomness of voting data, software NMR system requires a programmable voter. Our experiment using a simulated fault injector shows that voting software that executes on a processor has the time-of-check-to-time-of-use (TOCTTOU) vulnerabilities. In order to address the problem, we present a special-purpose voter processor and its embedded software architecture. The processor has a set of new instructions and hardware modules that are used by the software in order to accelerate the voting software execution and address the identified reliability problem. We have implemented the presented system on an FPGA platform. Our evaluation result shows that using the presented system reduces the execution time of error detection codes (commonly used in voting software) by 14% and their code size by 56%. Our fault injection experiments validate that the presented system removes the TOCTTOU vulnerabilities. This is achieved by using 0.7% extra hardware in a baseline processor. [34]

7.1. Motivation

Physical space exploration uses human or robotic spaceships to explore outer space (e.g., Moon, Mars, and beyond). As the exploration distance increases, autonomous operation and on-board data processing become an essential part of robotic spaceships. These operations are computation- and memory-intensive and require a high-performance, low-power computing platform. Thanks to the rapid advances in COTS devices, COTS-based

[34] An extended version of this chapter was published: K. S. Yim, V. Sidea, Z. Kalbarczyk, D. Chen, and R. K. Iyer, "A Fault-Tolerant Programmable Voter for Software-Based N-Modular Redundancy," in *Proceedings of the IEEE Aerospace Conference*, 2012.

platforms can satisfy these requirements with small design and development costs. The use of high-performance COTS chips in spaceships and other mission-critical systems, on the other hand, increases concerns about system reliability. This is due to high probabilities of transient faults as well as intermittent and permanent faults in these chips.

The use of high-performance COTS chips in spaceships, on the other hand, increases concerns about system reliability. This is due to high probabilities of transient faults as well as intermittent and permanent faults in these chips. A *transient fault* is caused by ionizing radiation, particle strike, or other external interference. The induced error can propagate and corrupt a software state but does not cause permanent damage to the hardware. In an experiment [CR11], injecting 50 MeV protons to three COTS devices caused data loss and device crash, but no symptom was observed that implied permanent damage of the devices (e.g., a high current condition). This type of faults is usually characterized as lasting up to one clock cycle. With the scaling of manufacturing technology, transients last longer because there is less space for the particle strike energy to dissipate. In current technology, transients can last up to two clock cycles or even longer [MM10]. If scaling continues, there is a high chance this duration will become longer. An *intermittent fault* has a duration that ranges between a few to billions of clock cycles [SSC03]. This fault is caused by various physical characteristics of chips (e.g., irregularities in chip voltage and temperature). Transistors become more and more susceptible to such physical characteristics with the scaling of manufacturing technology and the integration of many transistors into a chip (e.g., for multi-cores and large caches). A *permanent fault* occurs as a result of a manufacturing defect or excessive use of transistors. Such excessive use degrades the reliability of transistor materials. This fault behaves like a long duration intermittent fault, but it permanently compromises the functionality of the transistors. Both intermittent and permanent faults are more common in high-density chips because of their high utilization and low operating voltage.

One of the effective ways of synthesizing a reliable computer system from unreliable COTS components is *N*-tuple modular redundancy (NMR). NMR can be implemented in software without a major modification in the COTS components. For example, *N* identical copies of the same program can be run, with each copy executing on a separate computing module. Here, the time between two consecutive voting operations is generally longer than it is in hardware-implemented NMR. In order to reduce the voting interval time, software NMR can not only use the final program output data but also the intermediate program states as voting data. Intermediate states include large size data (e.g., vector or matrix of floating-point data). This makes hardware NMR voter unsuitable for

software NMR systems because hardware NMR voter uses only one bit or one word [SAC+99] as voting data.

In software NMR, the complexity of the voting algorithm is relatively high. If the voting data includes floating-point numbers, the voting algorithm should allow a certain degree of value errors because floating-point arithmetic is neither associative nor distributive. Also, voter data may include execution time and platform-dependent information that should not be directly compared with each other. Such a complex voting algorithm needs a programmable voter. However, developing a highly reliable programmable voter is challenging. A processor-based implementation (to offer programmability) generally has lower reliability than an ASIC-based implementation due to the large size of the hardware, high fault sensitivity (e.g., sequential circuits [SVK+05]), and the difficulty of verifying the design. This reliability problem of programmable voter is an important challenge because voter is the single point of failure.

In this chapter, we analyze the effectiveness of software-implemented error detection techniques under the presence of transient, intermittent, and permanent faults. We conduct fault injection experiments using a fault injector that is specifically developed to emulate errors induced by various types of faults in hardware. We observe the following problem:

- *Time-of-check-to-time-of-use (TOCTTOU) vulnerability.* Almost all software-based error detection codes check a system state before the state is used. If a hardware fault occurs after the checking (time-of-check) but before the checked data is used (time-of-use), this fault can evade the checking and harm the reliability or data integrity of voting software.

In this chapter, we present a fault-tolerant, programmable voter architecture for software NMR systems. The presented architecture consists of a special-purpose processor and its embedded software. The processor and software implement the following two main techniques:

- *Removal of the TOCTTOU vulnerabilities.* The TOCTTOU vulnerabilities are removed by novel instructions. These new instructions make the time-of-check and the time-of-use the same. Specifically, a target voting program is replicated, and both replicas execute on top of the special-purpose processor in sequence. The voting result data produced by these two replicas are compared and voted on using a new specialized instruction that does not suffer the TOCTTOU vulnerability.
- *Accelerating the execution of voting software.* New instructions are designed to speed up common software-based error detection codes. For example, a new instruction saves thread-dependent error checking data in special-purpose processor

registers. Another custom error checking instruction directly uses these saved data without re-computation or additional memory access. The saved data are copied to/from memory only during the context switching operations of user processes.

We have implemented a prototype on an FPGA platform by using a SPARC v8 processor as a baseline. The hardware area overhead of the presented architecture is 0.7% of the baseline processor size. The evaluation shows that using the presented instructions reduces the execution time of the five types of software-based error detection codes by 14 % and their code sizes by 56%, on average. Our fault injection-based validation experiment shows that the presented voter system does not have the TOCTTOU vulnerabilities.

The rest of this chapter is organized as follows. Section 7.2 describes the software NMR systems and their voter designs. Section 7.3 reviews the related works, and Section 7.4 analyzes the reliability problem of programmable voters. Section 7.5 presents our fault-tolerant voter processor and software with the three key techniques. Section 7.6 describes the prototype implementation. Section 7.7 performs validation experiments to analyze the provided error detection coverage. Section 7.8 evaluates the reductions in the performance and memory space overheads. Section 7.9 discusses other potential applications of the presented techniques. Section 7.10 reviews the related works, and Section 7.11 summarizes.

7.2. Background

This section describes background information of software NMR systems that can be built using COTS devices.

7.2.1. Software N-tuple Modular Redundancy

Space-borne software has strong reliability and availability requirements. Fault tolerance computers designed for such software use a high degree of redundancies in the hardware and/or software [SFA+07]. Software-based redundancy is preferred to hardware-based redundancy because that can be implemented by a system-level redesign (e.g., of the motherboard) without any intrusive chip-level modification.

Software NMR creates redundancies in space, time, or both dimensions. Figure 7.1 shows a spectrum of software NMR systems that use space-dimension redundancies where the systems are classified by the location of its voter software: (a) on top of a processor of a computer node that executes a replica, (b) on top of a separate chip or card in a compute node, or (c) on another node. The architecture (a) does not need an extra com-

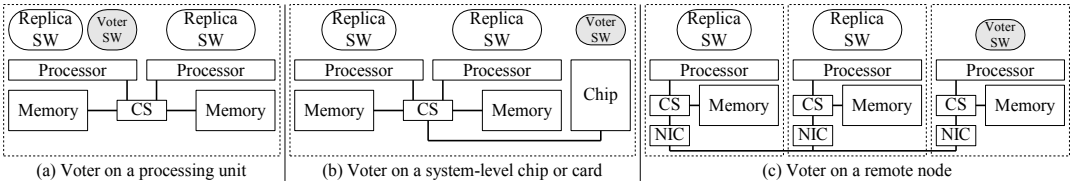


Figure 7.1. A spectrum of software-based NMR systems as a function of voter software location.

* SW is software, CS is chipset, and NIC is network interface card

puting module for the voter, however the execution of voting software perturbs the execution of the replica programs; this is especially critical if the replicas use frequently synchronized message-passing threads [TEF+05]. While the architectures in (a) and (b) cannot tolerate a node failure, the architecture in (c) can, but this costs an extra delay in the communication time between replicas and its voter.

The reliability of software NMR is lower than that of the simplex (i.e., executing only one program instance) after a certain time. We analyze the tradeoff point using field measurement failure rate data. The reliability of NMR is $R_v(t)\{1-(1-R_r(t))^n - n(1-R_r(t))^{n-1}R_r(t)\}$, where n is the number of replicas, $R_v(t)$ is the reliability of the voter, and $R_r(t)$ is the reliability of a replica. Let us assume that the replicas and voter follow the exponential failure rate distribution with the mean-time-to-failure (MTTF) of 1 and 100 time units, respectively, (i.e., $MTTF_r$=1, $MTTF_v$=100). The threshold point where the simplex configuration becomes more reliable is ~0.75 × $MTTF_r$ for TMR and ~2 × $MTTF_r$ for NMR (n=5). The time point at which the reliability becomes less than 0.99999 is ~0.00025 × $MTTF_r$ for TMR and ~0.001 × $MTTF_r$ for NMR (n=5). In order to calculate the actual $MTTF_r$ and $MTTF_v$ values, let us assume that the average failure rate of DRAM is about 25000 FIT (failure in time per billion device hours) per megabits [SPW09]. For one bit, this failure rate is 0.025 FIT (i.e., $MTTF_{bit}$ = 4.56 × 10^6 years). An SEC-DED ECC can correct one bit-error per data word, where c-bits are used as ECC bits to protect a data word of w-bits. MTTF of this ECC-protected memory system is $MTTF_{bit}/(w+c)\sqrt{\pi/(2M)}$, where M is the number of words in the memory [WQR+04]. Using this formula, MTTF of 1GB DRAM protected by an SEC-DED ECC with w = 32 and c = 7 is calculated as ~299.9 days.

This shows that when TMR is used, the voting interval shall be shorter than ~1.8 hours to provide 0.99999 of system reliability. Many target programs complete before this time interval if the program outputs are used for spacecraft control operations, for

example. For a target program that does not complete within this interval, the intermediate states of the program shall be checked by the voter at least at every ~1.8 hours.

7.2.2. Programmable Voter

Software NMR voter compares the intermediate states of replicas of a protected program if its voting interval (i.e., selected to provide a certain reliability) is shorter than the protected program execution time. In general, intermediate states have large size and are not exactly the same between replicas of the same program. A programmable voter (e.g., voting software on top of a processor) is more efficient for implementing checkers of intermediate states than a hardware voter [KS01]. This is because of the relative easiness of handling large volumes of voting data and customizing the voting algorithm to cope with the randomness in voting data when a programmable voter is used.

Hardware voter uses only one bit or one word [MM00] as voting data. Extending a hardware voter to check a large volume of intermediate data would significantly increase its design complexity and lower its reliability especially if the voting data has randomness as described follows:

(a) Errors in floating-point data. When target software is implemented using parallel threads, non-deterministic execution behavior of such parallel threads can make some portion of the program output data random. For example, if all parallel threads send floating-point data to a thread for summation, the arrival order of the sent floating-point data varies depending on various runtime conditions. Because floating-point addition is not associative, this can produce different sum values. This characteristic requires floating-point programs to allow for a certain amount of error in computation results (e.g., <0.1% value error [YPS+11]). Thus, a voter that checks whether two floating-point sum values are the exact same cannot find a matching condition even if the compared sums only differ by what is allowed by the target software output value error margin.

(b) Data that depend on runtime conditions. If the output data (or intermediate states) of replicas include some random data, the voter should exclude these data from the comparison. Since the presence and location of such random data in program outputs (or intermediate states) is not fixed in all software but depends on target software, a software voter can handle this more effectively than a hardware voter. Here, random data includes time information (e.g., date and time of execution) and platform information (e.g., process and node identifiers).

7.3. Classification

This section analyzes software-implemented error detection techniques used as part of a programmable voter. We classify the assembly code of an example software voter program and classify the existing software-implemented error detectors into five types.

7.3.1. Error Detectors

Let us assume a floating-point program that produces a matrix as the output. A possible voting algorithm is: checking the height and width of the output matrixes from different replicas, comparing the values of each matrix element (i.e., whether the absolute difference of values in the same location of the output matrixes is less than a threshold value), and checking whether all elements satisfy the compared condition (e.g., if one element does not satisfy the condition, the voting fails).

This simple voting algorithm uses three types of software-implemented error detectors that are: checking the equality (e.g., matrix height with a constant value), checking a value range (e.g., value difference is less than a threshold), and checking bitmasks (e.g., whether all comparison results are true if the results are stored as a stream of bits). More or less complex error detectors can be used, depending on the characteristics of the voting data and target software.

We analyze various types of software-implemented error detectors. Software error detectors typically insert a finite number of instructions and data variables into the source code or binary code of a target program. These inserted instructions and data perform error checking operations at runtime. We divide the inserted error checking instructions into four types:

(i) Creating redundancy. A certain type of redundancy in computation or data is introduced to detect an error induced by a hardware fault. Examples include duplication of instructions [RCV+05], memory data [DMZ09], or user processes [SMR+07]. Also, existing redundant hardware states can be directly or indirectly exploited to detect and/or fix the error (e.g., ECC for processor register or memory [SSP90]).

(ii) Checking the error condition. Error detection is done by checking the original and redundant states. These checking operations are classified into five subtypes (see the examples in Table 7.1).

- *Logical operation* checks a logical expression consisting of logical, arithmetic, and other operators (e.g., equality) [RCV+05][DMZ09][SMR+07][YPS+11].

Table 7.1. Example codes of common error checkers used in programmable voter (Notation: C/C++)

Type	Source Code Example
Logical condition	**if(v < 100 || v > 500)** Error(); *pointer = v; // data type of pointer is 'int *'
Bitmask	**if(v & mask != 0)** Error();
Vector	**for(int i=0; buffer[i] != '\0'; i++)** **if(i >= len)** Error();
Voting	int result = **(v1==v2 || v1==v3)?v1:((v2==v3)?v2:** Error()**);**
Table lookup	**if(table[(v - baseaddr) >> constant] & mask)** Error();

- *Bitmask operation* checks, for example, whether a set of bits are marked [CRA06].
- *Vector operation* checks a property of a vector (e.g., length) or values stored in a vector (e.g., value range).
- *Voting operation* checks whether a majority of values among the compared values are the same (e.g., 2 out of 3) [SMR+07][CRA06][YPS+11].
- *Table lookup operation* uses a lookup table that, for example, stores the access control information of a state that is being checked [ACR+08][CCM+09].

(iii) Invoking the error handler. If a checked error condition is met, a corresponding error handler is called. Methods to call an error handler include (a) procedure call, (b) system call, and (c) processor exception. While procedure call is the most common, a technique [Bag01] uses the divide-by-zero exception to avoid using an additional procedure call when checking equality.

(iv) Using the checked state. Checking is usually done before the checked state is used if the checker is embedded in the target software. This detects an error in a preemptive way (e.g., isolating an error to help error recovery) and takes advantage of the memory locality (i.e., similar effect as pre-fetching technique). Actual distance between checking and using is often determined by the error checker placement algorithm.

7.3.2. Example

Table 7.2 shows the implementation of an error detector that checks two logical conditions (see logical condition in Table 7.1). Assembly code in Table 7.2 is encoded using

Table 7.2. Assembly code implementation of an error checker that checks two logical conditions

Instruction sequence	Descriptions
cmp r1,99	cmp: compare instruction
ble t1	ble: branch less equal instruction
nop	nop: no operation instruction
cmp r1,500	r#: register whose number is #
bg t1	bg: branch greater instruction
nop	
b t2	b: unconditional branch instruction
nop	
t1: call Error	call: procedure call instruction
nop	
t2: st [r2], r1	st: memory store instruction

the SPARC v8 instruction set architecture (ISA)[35]. Although the checked conditions are simple, 10 instructions are added to implement the checking. Among the added instructions, the first 8 (bolded in Table 7.2) are always executing regardless of the presence of a hardware fault. Six of these 8 are for conditional branch operations.

A conditional branch operation is implemented as a pair of *cmp* (compare), b_{cond} (branch on a condition), and *nop* (no operation) instructions. Here, *nop* is used because SPARC v8 uses a delayed branch (a common technique in embedded processors).

Larger numbers of instructions are used to implement other error checkers. For example, implementation of a string length checker (see vector in Table 7.1) uses 21 instructions where some of these instructions manage the control-flow of a loop calculating the length of a null-terminated string. The error detector using a lookup table (see table lookup in Table 7.1) uses 17 instructions. Three instructions are used to convert a source address (*Src*) to an address (*TblTgt*) in the table as shown in (6.1). *SrcBase* is the source base address, *TblBase* is the table base address, and *SRL* (shift right logical) is a constant value chosen by the table mapping granularity. Five other instructions are used to implement a conditional branch operation that checks the value loaded from the table and calls an error handler when the condition is met.

$$TblTgt = TblBase + (Src - SrcBase) \gg SRL \tag{6.1}$$

[35]We use a SPARC v8 embedded processor as a baseline because this RISC processor has a relatively simple architecture to achieve a high reliability.

The overhead induced by these checkers in terms of the execution time and program code size depends not only on the number of instructions used to implement them but also on the frequency with which the checkers are used. Because the voter program is mainly used to compare data and to detect errors, a vast majority of voter program instructions is used to implement error checkers. This shows a need for accelerating the speed of error checking operations in a software voter.

7.4. Analysis

This section describes our fault injection experiment that analyzes the reliability of software error detectors frequently used in a programmable voter. Our analysis results show two types of their common vulnerabilities to hardware faults.

We analyze the behavior of a typical programmable voter under hardware faults as a function of the fault duration and fault location in the processor. This typical programmable voter is built by executing a simple voting program as a user process of an embedded Linux OS that runs on a SPARC v8 processor-based platform.

Observation 7.1: *Software-based error checkers used in programmable voters suffer from the TOCTTOU vulnerability. This can cause false positives and negatives in the voting results, where a false negative directly harms the availability and integrity of whole system.*

We analyze every execution cycle of the error detectors. Figure 7.2 shows the TOCTTOU vulnerability window of the error checker shown in Table 7.2 to a hardware fault. The depicted instruction sequence corresponds to a fault-free execution (i.e., checking two conditions and branching to *t2*) because the fault is not detected by the checker.

Three types of processor hardware faults can evade this error detector: (i) a hardware fault that changes the *reg1* value to a value smaller than or equal to 500 after the register access or A stage of the first *cmp* and before the execution or E stage of the second *cmp*, (ii) a hardware fault that changes the *reg1* value to any value after the A stage of the second *cmp* and before the E stage of the *st*, and (iii) a hardware fault that changes the fetched *reg1* value while the value is used by the *st*. (i) and (ii) correspond to the case when a fault is in the register file, and (iii) to when a fault is in the pipeline registers or combinational logics (see Figure 7.2).

The impact of a fault that evades the error detectors can be severe in a programmable voter. Let us assume the *r1* register in Table 7.2 contains the voting output data (e.g., the control signal for a navigation system of a spacecraft). If a corruption in the *r1* value is

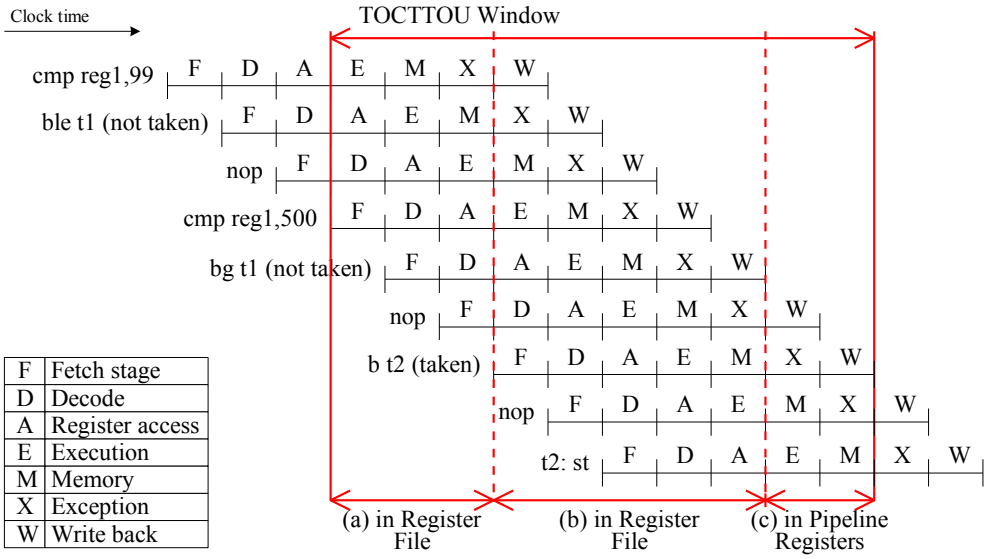


Figure 7.2. TOCTTOU windows of a software-implemented error detection code.

undetected, the data integrity of the whole system is broken and the reliability of whole system can be harmed. Similarly, when the corrupted data indicates which replica is faulty and needs a reboot, this can result in a continuous use of a faulty node or a restart of a non-faulty node that could cause severe damage, for example, when this node is currently the only non-faulty node.

The size of the TOCTTOU window is large when one of the following three conditions is present. *(i) The error checker is complex*. For example, an error detector checking multiple conditions has a larger TOCTTOU window than an error detector checking a single condition. *(ii) Multiple error checkers are integrated.* An example case is when both reliability and security checkers are used in the same place (e.g., [RCV+05] and [ACR+08] before every memory store). In this case, an error checker placed between the other error checker and the use instruction becomes a part of the TOCTTOU window of the other checker. *(iii) The checker is used frequently.* If the error detector is executed frequently, its cumulative TOCTTOU window size is large. For example, an error detector placed before every memory load has a larger cumulative TOCTTOU window than the same error detector placed before every system call.

7.5. Design

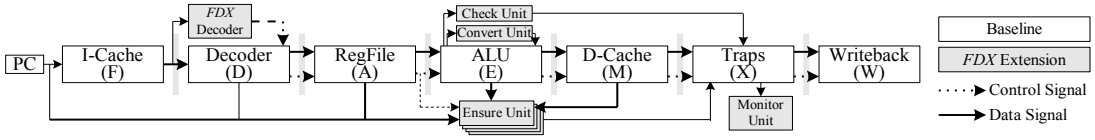


Figure 7.3. Architecture of the presented special-purpose processor for a software-implemented voter.

Instruction Syntax	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Description
(a) Check <SR1>,<Simm13>,<Cond> or <SR1>,<SR2>,<Cond>	Type (10)		0		Cond			Opcode (110111, CPOP2)						SR1					1	Simm13													Compares SR1 and SR2 and raises an exception if Cond is met.
																			0	0	0	Unused						SR2				
(b) Ensure <ID>,<SR1>,<TD>,<ST>	Type (10)		ID			ST: Target signal		Opcode (110110, CPOP1)						SR1: Source register					1	Co-opcode (00000, Ensure)					TD: Target distance								Configs an ensure unit (ID) so that it can ensure ST signal o TG instruction is same as SR1.
(c) Vote <Sr1>,<Sr2>,<Cond>, <ID>,<TD>, <ST>	Type (10)		0		Cond			Opcode (110111, CPOP2)						SR1					0	ID		TD: Target distance			TS: Target signal			SR2					Same as Check instruction. This configures a ensure unit (ID) using TD and TS.
(d) Convert <DR>,<SR1>,<I>	Type (10)		DR: Destination register					Opcode (110110, CPOP1)						SR1					0	Co-opcode (00001, Convert)					T: Base addr. type			Unused					Computes and stores {(SR1-SourceBaseReg) >> SRI + TableBaseReg to DR.

Figure 7.4. Encoding for new instructions designed on top of the SPARC v8 instruction set architecture.

This section presents the design of a fault-tolerant, programmable voter system for software NMR. The input of the voter system is *N* sets of data generated by *N* copies of the same software. This data is stored in a hardware queue that is protected by ECC. The output of the voter system is an *N*-bit word where each bit is the reset signal of each of the *N* replica nodes. It is assumed that all replica nodes have a bootstrap program that clones the states of other non-faulty replicas and continues the execution after rebooting.

Figure 7.3 shows the architecture of the presented special-purpose processor for the software-implemented voter. In the figure, white boxes are components in the baseline processor of SPARC v8, and gray boxes are added components. We name these additions the *Fault Detection eXtension (FDX)*.

Three types of units and one decoder module form *FDX*. These units are controllable by new instructions. Figure 7.4 summarizes the syntax and encoding of the new instructions designed for the presented processor. Instruction is used as a hardware-software communication interface. This is meant to alleviate the semantic gap between hardware and software (i.e., the difference in information available to hardware and software) in an efficient way as described follows.

7.5.1. Removing the TOCTTOU Window

This subsection describes techniques that are designed to remove the TOCTTOU vulnerability.

Figure 7.5 shows the execution model of voting programs on the presented special-purpose processor. Two copies of the same voting program are created and executed on

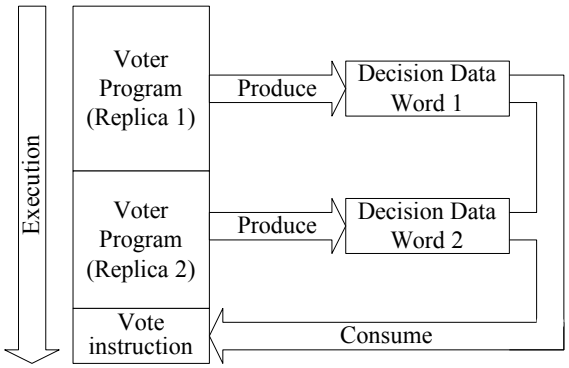


Figure 7.5. Presented voter software execution model.

the processor. Each program produces a data word that shows the voting results. For example, each bit in the data word can indicate whether a corresponding computing node is faulty and needs a restart. The two data words are compared and sent to a hardware unit, which resets specific computing nodes identified as faulty if the two data words are the same. For example, the two data words are different if one of the voting program copies experiences a hardware fault during its execution. The comparison operation of the two data words is not free from the TOCTTOU vulnerability insofar as this operation is implemented in software. This vulnerability is addressed using a new instruction (Vote), which is designed by merging two other new instructions (Ensure and Check).

7.5.1.1. Ensure Instruction

The *ensure instruction* is used to configure an ensure unit (see Figure 7.3) that monitors a specified processor signal. The instruction syntax is:

Ensure <SR>, [TD], [TS]

The ensure instruction reads the value of the source register <SR> and configures an ensure hardware unit. The configured ensure unit ensures that the <SR> value fetched by the ensure instruction is actually used by the target pipeline signal of a target instruction. Here, the target pipeline signal is specified by the target signal operand [TS], and the target instruction address (TA) that is calculated by adding the PC of the ensure instruction and the target distance [TD] value. If a different value is used by the target signal, an *FDX* exception is raised. In the syntax, the operand (e.g., [TD] or [TS]) represented in non-bolded text is directly encoded as a part of the instruction bitcode.

Table 7.3(b) shows an example code that uses the ensure instruction to eliminate the TOCTTOU window. In the original code in Table 7.3(a), a TOCTTOU window exists between the first *cmp* instruction and the *st* instruction. Thus, an ensure instruction is

Table 7.3. Optimizations of an error checking code by the presented instructions.

(a) Original	(b) Ensure	(c) Check	(d) Ensure/Check	(e) Vote
	Ensure r1,11,msd		Ensure r1,3,msd	Vote r1,99, le,2,msd
cmp r1,99	cmp r1,99	Check r1,99,le	Check r1,99,le	
ble t1	ble t1			
nop	nop			
cmp r1,500	cmp r1,500	Check r1,500,g	Check r1,500,g	Vote r1,500,g,1,msd
bg t1	bg t1			
nop	nop			
b t2	b t2			
nop	nop			
t1: call Error	t1: call Error			
nop	nop			
t2: st [r2], r1	t2: st [r2], r1	st [r2], r1	st [r2], r1	st [r2],r1

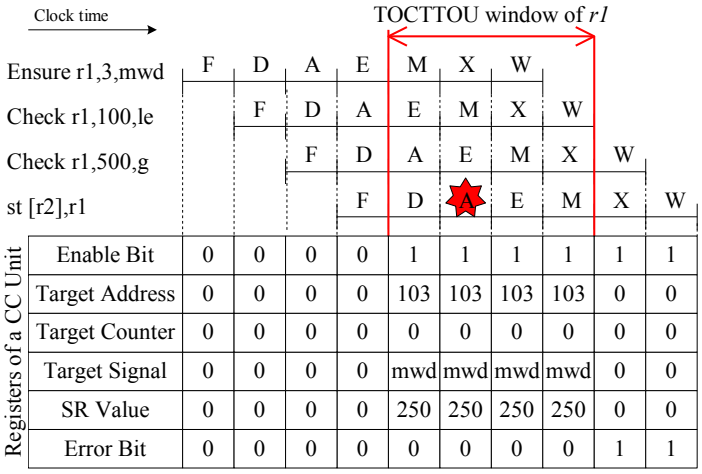
* cmp: compare, ble: branch less equal, nop: no operation, bg: branch greater, b: unconditional branch, call: procedure call, st: memory store, r#: register number #, msd: memory store data, le: less than, g: greater.

placed before the first *cmp* by using *r1* as <SR> (a register value being checked), $11_{(10)}$ as [TD] (as the *st* is 11 instructions away from this ensure instruction), and *msd* as [TS] (where *msd* means the target signal is memory store data). If a fault occurs in the *r1* register value after this ensure instruction and before executing the *st* instruction, this fault is detected by the configured ensure unit when the *st* is executing.

(i) Microarchitecture. To simultaneously monitor multiple target signals, multiple ensure units (e.g., 4) are provided. All ensure units monitor a control signal of the execute stage of the baseline pipeline. If the control signal indicating the ensure instruction is set, an idle ensure unit is activated. The activation is done in the memory stage of the ensure instruction (see Figure 7.6).

The activated ensure unit stores the value of source register <SR> and the [TS] value to its local registers. The target instruction address (TA) is calculated as described before and stored in a local register of the activated ensure unit. If [TS] is for a fetch stage signal, a counter is used to track the target instruction instead of TA because TA is ineffective for a fault in the fetched PC value. The counter value is initialized by the [TD] value. The enable bit of the activated ensure unit is finally set to complete this configuration.

Figure 7.6 shows how an ensure unit is configured and executed using the code in Table 7.3(d). After processing the execute stage of the check instruction (see Subsection 7.5.1.2), the enable bit of an idle check unit is set. At the same time, its target address register is set to $103_{(10)}$ by adding $100_{(10)}$ (the PC value of the ensure instruction in this example) and $3_{(10)}$ (the [TD] value). The given [TS] and read <SR> values (*msd* and 250, respectively) are set in the proper local registers. In the memory stage of the *st* instruction, this ensure unit reads the memory store data (*msd*) signal value and compares it to the SR register value. In this example, a soft error occurs in the register access stage of the *st* (see Figure 7.6) and changes the *r1* register value stored in the register file. Thus, a mis-



Enable Bit	0	0	0	0	1	1	1	1	1	1
Target Address	0	0	0	0	103	103	103	103	0	0
Target Counter	0	0	0	0	0	0	0	0	0	0
Target Signal	0	0	0	0	mwd	mwd	mwd	mwd	0	0
SR Value	0	0	0	0	250	250	250	250	0	0
Error Bit	0	0	0	0	0	0	0	0	1	1

Figure 7.6. Execution scenario of ensure instruction.

match is detected by this ensure unit. The error bit is set in the next pipeline cycle to raise an *FDX* exception. If a mismatch is not detected, the enable bit is cleared to reuse this ensure unit. Similarly, if the PC signal in the fetch stage is given as <TS>, the counter value of this ensure unit is decreased by one every time a new instruction is fetched. If the counter value is zero, the same concurrent check operation is conducted to check the data integrity of the target PC signal value.

(ii) Software. It is recommended that the ensure instruction be used just before a compare-equivalent instruction (e.g., *cmp* or check) to minimize the TOCTTOU window.

Four ensure units are used in our implementation because common error detectors in Table 7.1 check up to three states at the same place. These four ensure units are virtualized by the OS kernel in our implementation (or virtual machine monitor). At the context switch event, all internal registers of ensure units are treated as a part of the contexts of switched processes. Two other new instructions (see Section 7.5.2) are used to save and restore the ensure unit registers. This virtualization (or reconfiguration of the hardware checker) creates an illusion of multiple and exclusively accessible ensure units for each software execution entity (e.g., a user process).

7.5.1.2. Check Instruction

The *check instruction* supports common conditional branch operations. Its syntax is:

Check <SR1>, (<SR2> or [Simm]), [Cond]

The check instruction checks the contents of the source registers <SR1> and <SR2> for a logical condition [Cond] and raises an exception if the condition is met. Specifiable conditions are logical and bitmask operations. If an exception is raised, an error handler is

invoked by OS kernel or processor control logic. A signed immediate value [Simm] can replace <SR2>. Here, [Cond] and [Simm] values are provided as parts of the ensure instruction bitcode.

The check instruction is an optimization for the common control-flow path of conditional branch operations of error checkers. When there is no hardware fault, many conditional branch operations of error checkers take the same control-flow path. The execution of this common control-flow path is optimized by a check instruction, while supporting the other control-flow paths (the rare cases) is done using a processor exception.

Table 7.3(c) shows an example code of a check instruction. The original code shown in Table 7.3(a) has two conditional branch operations replaced by two check instructions. The two instructions (*call* and *nop*) in the original code are no longer needed because the error handler is directly called by the check instructions. This reduces the number of instructions used for error checking by 80% (from 10 to 2), and the number of executed instructions in the fault-free execution by 75% (from 8 to 2).

(i) Microarchitecture. Execution of the check instruction consists of the following three stages:

(a) Register access stage. <SR1> and <SR2> values are fetched from the register file (or the instruction register if [Simm] is used instead of <SR2>). If there is a dataflow dependency to values generated by the preceding instructions, these values are forwarded from the pipeline registers of the preceding instructions.

(b) Execute stage. An ALU in the baseline pipeline is used to compare the two fetched values and produce four flags: negative (N), zero (Z), overflow (V), and carry (C). These flags are forwarded to the check unit (see Figure 7.3) that performs a set of logical operations. For example, the equal-to condition is emulated by condition 'Z', less-than by 'Z OR (N XOR V)', and greater-than-or-equal-to by 'NOT (N XOR V)'. One of the 10 logical conditions is selected by [Cond] and a multiplexer in the check unit. Also, bitmask operations (mask-set and mask-unset) are supported. Mask-set checks whether all bits set in <SR2> are set in <SR1> (i.e., <SR1> AND <SR2> = <SR2>), where the AND operation is done by an ALU of the baseline pipeline and the equality is checked by the check unit.

(c) Exception stage. If the selected logical condition is met, an *FDX* exception is raised in this stage. For the *FDX* exception type, the precise interrupt mode is used that flushes all instructions fetched after the instruction that raised an exception. If an *FDX* exception is raised in a processor privileged mode, an *FDX* exception handler is called instead of stopping the processor (unlike other exceptions raised in the OS kernel).

A user-mode exception is used if the exception is raised in the user-mode and user-mode *FDX* exception is enabled for the current user-level process. This reduces the *FDX* exception handling overhead. Instead of trapping into the OS kernel, this user-mode exception looks up a user space exception vector table and branches to the corresponding exception handler. This is the key difference between the check instruction and the TWI (trap word immediate) or TI instruction in the POWER instruction set architecture, together with the more diverse conditions supported by the check instruction. Note that the user-mode exception enable bit is implemented by reusing a reserved bit in the processor status word. Thus, this bit is automatically treated as a part of the user process context in OS.

(ii) Software. Upon an *FDX* exception, the processor looks up the exception vector table in memory and jumps to a proper exception handler. For kernel-mode exceptions, an *FDX* exception handler is added in the OS kernel (SnapGear Linux for LEON3[36]). The *FDX* exception handler then calls an error handler defined in the kernel because this exception is raised in a privileged mode of processor.

For FDX exceptions raised in a user mode, a library of user-level programs that use this check instruction initializes the user exception handler register. The library stores the address of the user-level error recovery function to the user exception handler register. The library also enables the user-mode exception after initializing the register. If the user-mode exception is not enabled, the *FDX* exception handler in the OS kernel sends an *FDX* signal to the user process. The *FDX* signal information data contains the PC of an instruction raising this exception. A user process using *FDX* instructions is responsible for registering an *FDX* signal handler. If no user signal handler is given, the user process is terminated by default (fail-stop).

7.5.1.3. Vote Instruction

The *vote instruction* is designed to combine ensure and check instructions because these two instructions are typically used in sequence (see Table 7.3(d)). Its syntax is:

Vote <SR1>, <SR2>, [Cond], [TD], [TS]

The vote instruction is equivalent to the 'Ensure <SR1>, [TD], [TS]' and 'Check <SR1>, <SR2>, [Cond]' instruction sequence. Among these five operands, only two are fed by the register file (<SR1> and <SR2>). The rest are directly provided as parts of the vote instruction bitcode by using unused bits of the check instruction bitcode. Specifically, instead of supporting [Simm], vote instruction uses a part of [Simm] encoding bits for

[36]LEON3 Processor, http://www.gaisler.com

[TD] and [TS] operands (see Figure 7.4). Because of this limit in encoding space, [TD] and [TS] operands of the vote instruction support only a subset of what [TD] and [TS] operands of the check instruction can represent (i.e., common cases are chosen as the subset).

The vote instruction further reduces the code size and execution time of error checking while removing the TOCTTOU window. An example code shown in Table 7.3(d) is reduced to two instructions by using two vote instructions (see Table 7.3(e)). This is a 33 % reduction (from 3 to 2 instructions) in the code size, and the same reduction in the executed instruction count in the fault-free case.

7.5.2. Accelerating Error Checking

This subsection presents instructions that can accelerate the error checking operations.

7.5.2.1. Convert Instruction

The *convert instruction* accelerates the address conversion of the table lookup operation (see Table 7.3). Its syntax is:

Convert <DR>, <SR>, [Type]

The convert instruction converts the source address <SR> to the address in the lookup table using a formula equivalent to (7.1) and stores the converted address to the destination register <DR> where [Type] selects one of preconfigured address conversion rules.

(i) Microarchitecture. To convert an address in a cycle, Equation (7.2), which is analogous to (6.1), is used for the conversion operation. If a conversion rule is fixed in application software, *SrcBase*, *TblBase*, and *SRL* are all constant values. Thus, the (*SRL*) term and the (*TblBase*-(*SrcBase*>>*SRL*)) term are computed and stored in two internal C1 and C2 registers of the convert unit (see Figure 7.3).

$$\mathrm{A_{TblTgt}} = (\mathrm{A_{Src}} \gg \boxed{\mathrm{SRL}}) + \boxed{\mathrm{A_{TblBase}} - (\mathrm{A_{SrcBase}} \gg \mathrm{SRL})} \qquad (7.2)$$

Multiple sets of C1 and C2 registers are used, where one of them is selected by [Type]. In the execute stage of the convert instruction, a logical shift right and an addition operation are performed by the convert unit using the selected C1 and C2 values.

(ii) Software. The internal C1 and C2 registers of the convert unit are programmable by user mode software. For example, the *main()* function of a user program can be instrumented to pre-compute C1 and C2 register values and store them to a set of C1 and C2 registers of the convert unit by using new instructions (see Subsection 7.5.3.2). This dynamic pre-computation is correct even when the base address of the lookup table is dynamically chosen (e.g., due to the use of address space layout randomization [SPP+04]).

Multiple sets of C1 and C2 registers are programmed if a target program uses multiple lookup tables for its error checking operations. These C1 and C2 registers are virtualized by treating them as a part of the target program context as explained before.

7.5.2.2. Configuration Load and Store Instructions

The configuration load (CLD) *and store* (CST) *instructions* are used to access the configuration where configuration refers to the internal registers of the *FDX* hardware units. Their syntaxes are:

CLD [SA], <DR> | CST <SR>, [DA]

The CLD instruction loads configuration data specified by the source address operand [SA] to a processor pipeline register <DR>. The CST instruction stores the value of a pipeline register <SR> to a configuration register specified by the destination address operand [DA]. Accesses to some configuration registers are controlled by the *FDX* decoder (see Figure 7.3). For example, direct write access to internal registers of ensure units is allowed only if the processor is in a privileged mode unlike to the C1 and C2 registers.

7.6. Implementation

This section describes our implementation for proof-of-concept of the presented system.

7.6.1. Hardware Area Overhead

We have implemented the presented *FDX* on an FPGA-based platform (Altera Stratix-II) by extending a SPARC v8 processor. The *FDX* hardware includes an *FDX* decoder, four ensure units, a check unit, a convert unit, a monitor unit, and additional pipeline control and data registers (see Figure 7.3). Hardware area overhead is 1.11% for each ensure unit, 0.04% for the check unit, 0.14% for the convert unit, 0.69% for the monitor unit, and 0.86% for all the rest (the *FDX* decoder and pipeline register extension). These ratios are normalized to the area size of the baseline processor core (excluding cache, TLB, and memory controller), where area sizes are calculated using gate counts of the synthesized hardware. The total area overhead of the *FDX* extension is 6.17% compared with the baseline, single-issue, in-order core size, and it is 0.7% compared with total baseline processor size.

Adding *FDX* does not delay the clock cycle of the baseline processor when the baseline clock frequency is 80MHz on the used FPGA. This is because *FDX* is simpler than the logics and data paths of the corresponding pipeline stages of the baseline processor.

Table 7.4. Example two types of macro functions to encode the check instruction (C/C++ SPARC inline assembly)

(a)

```
#define FDX_CHECK(sr1id,sr2id,cond) \
    __asm__(".word 0x81b80000 + ((" cond " & 0xf) << 25) + (((" sr1id ")
            & 0x1f) << 14) + ((" sr2id ") & 0x1f)")
```

(b)

```
#define FDX_CHECK_EQ(sr1,sr2) \
    __asm__("cpop2 [%0 + %1], %%g1\n\t" :: "r"(sr1), "r"(sr2));
```

7.6.2. Programming Interface

We describe the software extensions that offer a programming interface (i.e., macros) to the new instructions.

Error detectors written by the programmer can be rewritten using the *FDX* macro functions, which are implemented as a C library using inline assembly code (see Table 7.4). If the physical register identifiers of operand variables are known, the macro shown in Table 7.4(a) is used for check instructions where *sr1id* and *sr2id* parameters contain the physical register identifiers of <SR1> and <SR2> operands, and *cond* contains the bit pattern of the [Cond] operand (e.g., '0x1' for equal). The physical register identifier can be derived by disassembling the binary file. If the binary file already contains and uses this macro, changing the parameter value of the macro does not change the physical register identifiers after a re-compilation.

To automate this physical register identifier derivation process, we design the other type of macro as shown in Table 7.4(b). This type of macros accepts program variables as parameters (e.g., *sr1*, *sr2*) and sends them to the inline assembler that generates the physical register identifiers of these variables. In this example, because the check instruction uses the DR (destination register) field of original coprocessor instruction as the [Cond] field, multiple check instruction macros are designed such that each uses a different constant value for the [Cond] field (in Table 7.4(b), *%%g1* is to specify the value of 1 for equal conditions). These two types of macros are tested with a GCC cross compiler v3.4.4 for SPARC v8 ISA.

Various types of error detection libraries can be designed using these macros. For example, *assert()* macros can be rewritten to use the check or vote instruction. These library functions make migration of complex voting programs to the presented architecture easy. If error detectors are already implemented as a library and no change is made in their interface to applications after using the *FDX* instructions, the use of *FDX* is transparent to target application software.

When using multiple ensure instructions, it is recommended to minimize overlaps between delayed checking intervals (i.e., from an ensure instruction to its target instruction). Ensure instruction intervals can safely contain a control-flow instruction if the target of control-flow instruction is analyzed at compile-time and the target does not use more ensure units than available. This does not mean the ensure instruction interval cannot contain any control-flow instruction. A common use of an ensure instruction is to monitor the target address of a control-flow instruction (e.g., a function entry). By calculating a distance to the control-flow instruction, using this distance +1 as [TA], and specifying fetch-PC as target signal (TS), the ensure instruction can check whether the control-flow is properly made.

7.7. Validation

This section analyzes error detection coverage of systems with and without *FDX*.

7.7.1. Validating Error Detection of FDX-Based System

We validate the error detection coverage of voter software implemented using the *FDX* instructions. A register transfer level (RTL) simulation is used to validate that the identified TOCTTOU windows are removed when the *FDX* instructions are properly used. RTL simulation allows us to inject a hardware fault and monitor the consequences without causing any interference to the execution behavior of the target system pipeline.

A voter system that follows the *FDX* software architecture executes the same voting program twice in sequence. Each of the voting program executions saves its voting result data (e.g., 32-bits) to memory. After the executions, the voter software uses memory load instructions to read the voting result data from the memory to the processor registers (e.g., *r1* and *r2*) and compares the read data. If they are the same, the voting software writes the value of *r1* to a memory-mapped I/O address in order to output the voting result data (e.g., that resets some replica nodes or continues without any reset).

If a fault occurs in the execution of one of the two voting programs, this either causes two different voting result data or leads to an observable failure (e.g., crash or hang). If there is a corruption in the voting result data, the comparison instruction detects this at the end of voting software execution. If there is an observable failure, this is detectable and tolerable by an existing technique (e.g., watchdog-based reboot).

We thus focus on analyzing the error detection coverage of the last comparison and memory write instructions of the *FDX*-based voting software. We consider two types of

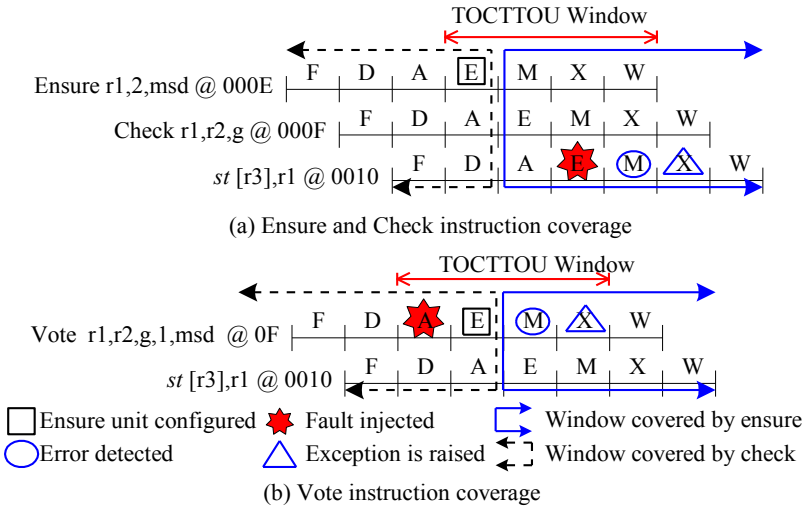


Figure 7.7. Error detection coverage of *FDX* instructions.

software. One uses the ensure and check instructions (see Figure 7.7(a)). The other uses the vote instruction (see Figure 7.7(b)). In both cases, *r1* and *r2* contain the voting result data, and *r3* contains the memory-mapped I/O address where the voting result data is sent to. For the instruction sequences given in Figure 7.7, a fault is emulated in each pipeline stage from the fetch cycle of the first instruction (ensure in Figure 7.7(a) and vote in Figure 7.7(b)) to the memory stage of the last instruction (*st*).

(a) Ensure instruction. The ensure instruction detects many faults injected in the TOCTTOU window. In Figure 7.7(a), the error detector code has the ensure, check, and *st* instruction sequence. For example, let us consider a hardware fault emulated in the execute stage of the *st* instruction to change the pipeline register containing memory store data (*msd*) value. This induced error is detected by an ensure unit in the memory stage of the *st*. This raises an *FDX* exception in the next pipeline cycle. Similarly, the ensure instruction detects all faults injected in the target state after its execute stage (see the window covered by ensure in Figure 7.7(a)).

(b) Check instruction. The ensure instruction does not detect faults that occur before its memory stage (e.g., execute stage) because an ensure unit is enabled from the memory stage. As recommended, if a compare-equivalent instruction follows an ensure instruction, the following instruction can detect such uncovered faults of the ensure instruction. For example, if a fault occurs in *r1* stored in register file at the register access stage of the check instruction (in Figure 7.7(a)), the corrupted *r1* value is compared to *r1* in its execute stage. If the corrupted *r1* value is bigger than *r2*, this check instruction raises an

FDX exception in this example. This shows that the check instruction can detect faults that occur before the memory stage of its preceding ensure instruction (see the window covered by check in Figure 7.7(a)).

(c) Vote instruction. Figure 7.7(b) shows the coverage of the vote instruction. This coverage combines the coverage of the ensure and check instructions. The vote instruction is more effective than a pair of ensure and check instructions if the execute stage of the ensure instruction is processed before the register access stage of the check instruction (e.g., due to a delay caused to forward the *r2* value to the check).

One may consider the use of a compiler-based technique to remove the TOCTTOU window by reordering time-of-check and time-of-use. This is not applicable when the use is a control-flow instruction because a fault can change the control-flow (bypassing the checker placed after the use instruction). Also, this cannot detect a fault if the induced error does not remain in the processor (e.g., after propagated to the memory). For example, if a fault occurs in a pipeline register, the following instruction cannot see the corruption unless the corrupted pipeline register value is forwarded or stored in the register file.

7.7.2. TOCTTOU Window Size in Non-FDX-Based System

We then evaluate the ratio of random hardware faults that evade a given software-based error detector by exploiting a TOCTTOU vulnerable window. The probability of missing a hardware fault due to this vulnerability depends on the relative distance between check and use, and on the size of the checked state where there parameters are determined by the type of used error detectors. We use one of the strongest software-implemented fault detection techniques (i.e., instruction duplication [RCV+05]) by manually instrumenting the assembly codes of two benchmark programs: a Hanoi tower, representing an integer program; and a matrix multiplication, representing a floating-point program. Note that the claimed error detection coverage of this technique in [RCV+05] was 100% because the fault injection was conducted in the processor architecture layer.

We conduct a fault injection experiment in the microarchitecture layer. We collect 25,152 fault injection samples on small pieces of the target program code. Despite of the use of one of the strongest software-implemented error detectors, on average, 0.56% of the faults injected in the processor microarchitecture states evade the error detector and lead to silent data corruption partially due to TOCTTOU windows.

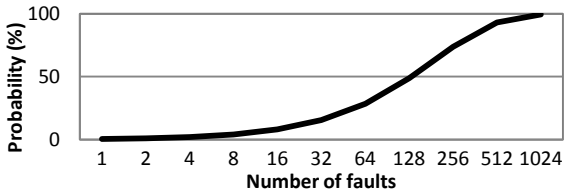


Figure 7.8. Probability of at least one undetected fault due to the TOCTTOU vulnerability.

* This is when a program experiences multiple faults (x-axis is fault count) over its lifetime.

Figure 7.8 analyzes the probability of undetected hardware fault due to the TOCTTOU vulnerability when a program experiences multiple faults over its lifetime. This data is calculated using a formula, $1-(1-P)^N$, where P is the probability of an undetected fault due to the TOCTTOU vulnerability (0.56% obtained in our experiment) and N is the number of faults (the x-axis in Figure 7.8). As the program experiences more hardware faults, the probability of missing at least one fault becomes higher (e.g., 99.5% when the fault count is 1,024). A program can experience many faults when (a) its execution time is long (e.g., scientific programs) [YKI09] or (b) the hardware fault rate is high. In either of these cases, even this small TOCTTOU vulnerability window can seriously harm the reliability and data integrity of target program protected by using a strong software-based fault detection technique. This shows the importance of using *FDX*-like extra fault detection in mission-critical systems where near perfect error detection coverage is needed because of the possibility of catastrophic failures when the system operates with an undetected error.

7.8. Evaluation

This section evaluates the reductions in the performance and code size overheads when *FDX* is used.

7.8.1. Performance Overhead Reduction

We measure the execution time of the five error detectors (listed in Table 7.1) while using different *FDX* instruction sets. We use a real-time signal tracing tool (SignalTap II) to measure the execution time at the cycle granularity level. The time from the first cycle of the first instruction of an error checker to the last cycle of the last instruction of the checker is measured. To accurately measure the computation time, the same error checker

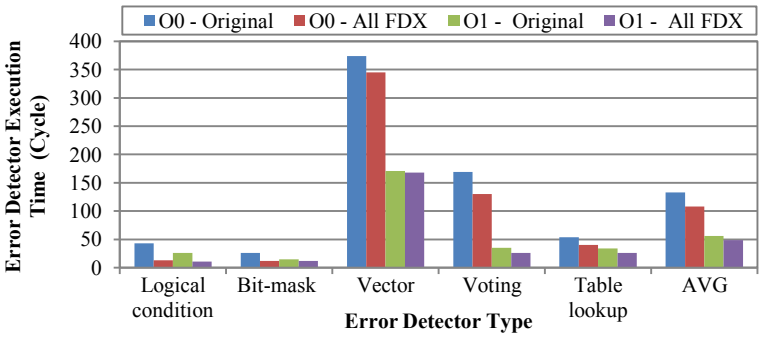


Figure 7.9. Error detector execution time reduction as a function of the used compiler optimization level (*O0* or *O1*) and instruction sets.

is executed once before measurement to reduce the measurement interferences (e.g., caused by cache misses and page faults).

On average, using all *FDX* instructions reduces the execution time of error detectors by 18.9% when the compiler optimization is not used and by 13.5% when the compiler optimization is used (see AVG bars in Figure 7.9). This ratio is normalized to the execution time of assembly codes of error detectors generated for the original ISA using the same compiler optimization level. Much higher speedup ratios are observed if the error checking algorithm is simple. For example, the execution time of the logical condition error checker (with optimization) is improved by 57.7% if all *FDX* instructions are used because of its simple checking algorithm, i.e., checking two logical conditions.

The average execution time reduction by *FDX* is smaller when compiler optimization is used. For example, compiler optimization combines two conditional branch operations into one operation (see the original code in Table 7.1). The optimized code adds -100 to *r1* and uses a branch instruction (*bleu: branch on less-than-equal unsigned*) to change the control flow when the *r1* value is less than or equal to 400. Because *r1* is treated as an unsigned integer value, if the original value of *r1* is smaller than 100, the *r1* value becomes a big positive integer number after the addition. In this case, the *bleu* instruction is not taken, and thus the optimized code executes the next instruction and calls an error handler. Only the pair of *cmp* and *bleu* instructions are replaced by a check instruction. Compiler optimization can also remove some *cmp* instructions by using an arithmetic instruction that generates flags (e.g., *addcc/btst*). In addition, while un-optimized codes do not use the delayed branch slot, optimized codes do so when it is feasible.

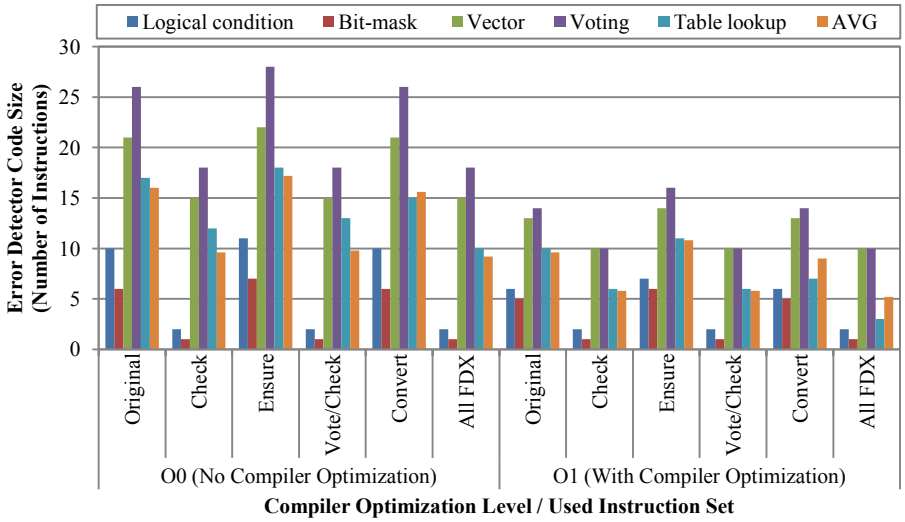


Figure 7.10. Error detector code size reduction.

7.8.2. Code Size Overhead Reduction

Figure 7.10 shows the program code size of five types of error detectors (listed in Table 7.1) as a function of the used *FDX* instructions and compiler optimization level. A GCC v3.4.4 for SPARC ISA is used as a C/C++ compiler. We consider two optimization levels (O0 and O1) because optimization levels higher than O1 increase the code size in some types of checkers.

The average code size reduction is 55.8% when all *FDX* instructions are used and compiler optimization is enabled (O1) (see AVG bars in Figure 7.10). This reduction ratio is normalized to the code size of error checkers generated for the original ISA using the compiler optimization (O1). If all *FDX* instructions are applied to the checker codes compiled without optimization (O0), the average code size reduction ratio is 52.8% as compared with the size of the un-optimized original code (O0).

All *FDX* instructions except for the ensure instruction reduce the code size. Specifically, the check and vote instructions are effective for all types of error checkers, while the convert instruction is only useful for an error checker using a lookup table.

Table 7.5 shows the assembly codes for common error detection routines. Note that these codes are derived from existing error and attack detection techniques and are different from the codes in Table 7.1. Here, each code block checks a condition (above a dotted line) and two conditions (below a dotted line). The left columns are the original code. The right column of each type of error detector codes is a code that is the same as one in the left column but is transformed by using the *FDX* instructions.

Table 7.5. Software-implemented hardware fault detectors and examples of using *FDX* instructions.

(a) Logical condition

MUL g1,Const,g1	Ensure g1,3,msa
CMP g1,g2	MUL g1,Const,g1
BLT ERROR	Check g1,g2,ne
NOP	
ST [g1],g0	ST [g1],g0
- - - - - - -	- - - - - - - -
MUL g1,Const1,g1	MUL g1,Const1,g1
CMP g1,g2	Vote g1,Const1,ne 2,msa
BNE ERROR	
NOP	Check g1,Const2,lt
CMP g1,Const2	
BLT ERROR	
NOP	
ST [g1],g0	ST [g1],g0

(b) Mask

AND g2,Mask,g1	Vote g1,Mask,ms,1,msa
CMP g2,Mask	
BNE ERROR	
NOP	
ST [g1],g0	
- - - - - - -	ST [g1]←g0
AND g2,Mask1,g1	- - - - - - - -
CMP g2,Mask1	Ensure g1,7,msa
BE P1	AND g2←Mask1,g1
NOP	CMP g2,Mask1
AND g2,Mask2,g1	BE P1
CMP g2,Mask2	NOP
BNE ERROR	Check g1,Mask2,ms
NOP	
P1: ST [g1],g0	P1: ST [g1]←g0

(c) Duplication

CMP g1,g2	Vote g1,g2,ne,1,msa
BNE ERROR	
NOP	
ST [g1],g0	ST [g1],g0
- - - - - -	- - - - - - - -
CMP g1,g2	Vote g1,g2,ne,2,msa
BNE ERROR	
NOP	
CMP g3,g4	Vote g3,g4,ne,1,msd
BNE ERROR	
NOP	
ST [g1],g3	ST [g1],g3

(d) Value range

CMP g1,g2	Vote g1,g2,lt,2,msa
BLT ERROR	
NOP	
CMP g1,g3	Check g1,g3,gt
BGT ERROR	
NOP	
ST [g1],g0	ST [g1],g0
- - - - - -	- - - - - - - -
CMP g1,g2	Vote g1,g2,lt,4,msa
BLT ERROR	
NOP	
CMP g1,g5	Check g1,g5,gt
BGT ERROR	
NOP	
CMP g1,g3	Check g1,g3,gt
BGT ERROR	
NOP	
CMP g1,g4	Check g1,g4,lt
BLT ERROR	
NOP	
ST [g1],g0	ST [g1],g0

(e) Lookup table

SUB g2,g1,MIN	Ensure g1,5,msa
DIV g2,g2,8	Convert g2,g1,0
ADD g2,g2,Base	LD g3,[g2]
LD g3,[g2]	Check g3,Color,NE
CMP g3,Color	
BNE ERROR	
NOP	
ST [g1],g0	ST [g1],g0
- - - - - - - -	- - - - - - - -
SUB g2,g1,MIN	Ensure g1,8,msa
DIV g2,g2,8	Convert g2,g1,0
ADD g2,g2,Base	LD g3,[g2]
LD g3,[g2]	AND g4,g3,Mask1
AND g4,g3,Mask1	CMP g4,Mask1
CMP g4,Mask1	BE P1
BE P1	NOP
NOP	Check g3,Mask2,and
AND g4,g3,Mask2	
CMP g4,Mask2	
BNE ERROR	P1: ST [g1],g0
NOP	
P1: ST [g1],g0	

* AND: Logical AND, BE: Branch equal, BGT: Branch greater than, BLT: Branch less than, BNE: Branch not equal, CMP: Compare, DIV: Divide, LD: Memory load, MUL: Multiply, NOP: No operation, ST: Memory store, SUB: Subtract, g0-5: Processor registers, [g0-5]: Memory pointer, msa: memory store address, msd: memory store data, ms: maskset, lt: less than., ne: negative equal, gt: greater than

* Ensure, check, vote, and convert instructions (see Figure 7)

(a) Logical condition. An error detector checks whether one or multiple logical conditions are met. An *assert()* statement (e.g., var1 == var2, var1 < var2) in a C program is a representative example. In Table 7.5(a), the code checks a condition (g1 × Const = g2, same as the TRUMP [CRA06]) and two conditions (g1 = g2 and g1 ≥ Const2). Each of these condition checks is implemented by a vote instruction.

(b) Bitmask. Mask is another common error checker, which can, for example, check whether a certain set of bits are set or unset. The code in Table 7.5(b) checks whether all bits set in the Mask variable are set (same as MASK [CRA06] and the call/return target checking in Native Client [YSD+09]). This can also be efficiently optimized using the check or vote instruction because the check unit supports these bitmask operations. The code in Table 7.5(b) then checks whether all bits set in Mask1 or Mask2 are set. Here, if one condition is true, the result is true (ORed). *FDX* cannot replace the first conditional

branch (after the dotted line in Table 7.5(b)) with a check or vote instruction. This is because this first branch bypasses the rest of the error check if the checked condition is met, while check and vote instructions can be used only if the target of a control-flow instruction is an error handler.

(c) Duplication. Checkers for duplication-based error detectors usually use either equal or not-equal operator (e.g., EDDI/SWIFT [RCV+05][CRA06][OSM02]). This typically requires all duplicated states to be the same (i.e., conditions are ANDed if multiple conditions are checked). In this case, after the first check, the checker can jump to the error handler if that condition is false, but it still needs to process the rest of conditions if that condition is true (see Table 7.5(c)). This pattern is common in SWIFT, for example, because it checks two conditions (for memory address and data operands) before each memory write instruction. In SWIFT-R [CRA06], computation results are triplicated and thus two comparison operators are used in the same place to check whether all three results are the same. These equal or not-equal checkers can be optimized using the vote instruction, as shown in Table 7.5(c).

(d) Value range. Checking whether a target value is within a value range is another common method of error detection. Examples are data value checkers [SLR+08] (e.g., for input validation) and memory segment checkers [YSD+09]. In Table 7.5(d), the code checks a condition ($g1 \leq g3$) and two conditions ($g2 \leq g1 \leq g3$ or $g4 \leq g1 \leq g5$). As this check operation is a subset of logical conditions, this type of error detection is optimized using *FDX* instructions.

(e) Lookup table. Applications of the *FDX* instructions are limited to hardware fault detection. A look-up table is most commonly used to detect and isolate software faults and security bugs [ACR+08][CCM+09][YSD+09]. For example, WIT checks whether pointer-based memory write instructions are accessing only data they are supposed to access. An access permission table is used to specify which memory write instructions have access permission to each portion (e.g., 8 bytes) of data memory. This permission table is updated right after every memory object allocation and deallocation (e.g., by *malloc()* and *free()*). This table is also read right before every pointer-based memory write. As exemplified in Table 7.5(e), this operation is well supported by the convert and check instructions.

7.9. Discussion

This section discusses other potential uses of the presented processor.

N-version programming (NVP) is a software engineering process that can reduce the probability of concurrent replica failures [Avi85]. In order to reduce such probability, NVP uses different types of software and/or hardware platforms for its replicas. Its replicas on different types of platforms are more likely to have different intermediate data values and produce different output data values (e.g., if floating-point operation is used) than replicas on the same type of platforms (i.e., in software NMR). Such characteristic requires a programmable voter, and consequently makes the presented voter more attractive to NVP systems.

The presented techniques can improve the error detection and recovery coverage of computing nodes. For example, embedding software-implemented error checkers in target software can detect errors in a computing node and send this information to an external voter. This eventually improves the coverage of NMR systems especially when N is small. For example, in a TMR system, at least two replicas must be operating correctly in order to detect and tolerate faulty replicas. Because the time before voting is relatively long in software-based TMR, the simultaneous failure probability of two replicas (e.g., common mode failure) is relatively high. If the error detection information of each replica is available and trustworthy, the voter can find the non-corrupted third replica and tolerate the concurrent failure of two replicas.

The use of software-implemented error detection in target software allows replicas to preemptively detect an error. This prevents the propagation of the error to another state (e.g., outside of target node). Such preemptive detection not only reduces the probability of common mode failures but also makes it possible to restart the faulty computing node in advance without having to wait until the next voting operation time. Consequently, we believe that the presented techniques are generally useful to detect and tolerate errors in embedded processors.

7.10. Related Work

This section reviews the related work. The existing error detection and recovery techniques are classified into six types:

(i) Hardware-based redundancy. Redundancy is introduced in various types of hardware layers. These layers include the transistor, circuit, microarchitecture, and architecture layers of the chip (e.g., Razor [EKD+03]). In an architecture-level technique, the entire processor pipeline is replicated to compare the execution results of every instruction. If a mismatch is detected, without committing the execution result, the instruction is re-

executed. The G5 processor [27] is an example design for in-order processor pipeline architecture. Thanks to its fine-grained checking, this technique offers high error detection and recovery coverage for duplicated components and short detection and recovery times.

Designing a hardware redundancy system for a high-performance computer is costly because of the use of many cores and sophisticated core architectures (e.g., out-of-order execution and branch prediction). For example, for out-of-order processor architecture, a proposal (DIVA [Aus99]) exists, but no chip has been fabricated yet for commercial purposes.

When many hardware replicas are used, the complexity of the system and the voter grow. This reduces the reliability of the voter. A previous work [SFA+07] reduces the development cost of NMR computers, except for the voter, by modularizing computing elements. Note also that the use of NMR does not provide perfect fault tolerance coverage if the target software runs on top of multiple processors or nodes, which can result in the use of an extra upper-layer fault tolerance technique in practice.

(ii) Hardware-based customized protection. Symptom-based techniques (e.g., ReStore [WP05] and SWAT [LRS+08]) optimize error detection for general error propagation paths of software. In a more aggressive approach (e.g., RSE [NKI+04]), signatures of each application software are programmed in hardware when the software is loaded. The programmed hardware concurrently monitors the execution behavior of the application. This hardware detector targets non-benign errors in the application software. However, programming a large volume of application signatures makes it less flexible, especially if it is used to monitor multiple applications.

This shows the difficulty of developing hardware-software co-designed error detection techniques and the importance of the hardware-software interface in co-design. Because of the semantic information gap between software- and hardware-based error checking mechanisms, to improve the detection coverage, the error sensitivity of application states can be analyzed by software and delivered to hardware in a timely way. While in some previous work, all information is delivered when the program is loaded, in our design only a necessary portion of this information is delivered using the standard hardware-software interface (e.g., before a target state is used). This takes advantage of memory locality and hardware parallelism.

(iii) Application-specific instruction set processor (ASIP). ASIP is extensively used for domain-specific applications (e.g., digital signal processing and reconfigurable computing), where the general-purpose instruction set is extended by extra, customized instructions. This allows the designer to maximize the benefit vs. cost, e.g., by replacing frequently used instruction sequences with simple hardware implementation [CFH+04].

Many previous ASIP designs are done for performance not dependability, while [KNS10] is done for error recovery, and [RPK05] is done for security. This is because of the difficulties of quantifying reliability as a function of ISA design and customizing an instruction sequence containing control-flow instructions. As far as we know, this work is the first ASIP design for fault detection.

(iv) Complex instructions. Some processors have instructions that can be used to optimize error detector implementation. For example, predicate instructions in ARM ISA are conditionally executed and thus can remove a branch instruction in error checking. This execution has a larger overhead than the check instruction because the predicate instructions are always fetched and decoded regardless of the program control flow. Also, in a superscalar processor, speculation and branch prediction are used to hide the control hazard (a delay due to a branch instruction) although the hardware area overheads (e.g., for branch target buffer) are not negligible. In x86 ISA, index instructions are supported that can also accelerate the address conversion operation of the table lookup.

(v) Instruction scheduling. When an error detector is derived by a compiler technique, the derived error detection routines can be fully customized for processor ISA. For example, BGI [CCM+09] uses a table lookup operation for software fault isolation and error detection. By fixing the lookup table location in a virtual address, BGI simplifies the address conversion operation to two instructions on a complex instruction set computer machine (*sar* and *btc* in x86 ISA). This, however, is not compatible with some other techniques (e.g., address space layout randomization) that dynamically set the lookup table base address to protect the table from malicious and privileged users.

Many compiler-generated error detectors optimize the redundancy creation stage (see Section 7.2) by exploiting hardware parallelism [RCV+05][DMZ09][SMR+07]. In this chapter, we optimize two of the remaining three stages of error detectors (checking a condition and calling an error handler), which implies that the presented techniques are orthogonal to many previous techniques.

(vi) Built-In Self-Test (BIST). BIST is an efficient technique to diagnose permanent faults. On the other hand, BIST is impractical for diagnosing intermittent or transient faults due to its separate execution mode. Mode switching is needed if the processor is on and executing software.

7.11. Summary

A failure in mission-critical system is either catastrophic or expensive to fix. In order to analyze the reliability of such systems, we have built a simulated fault injector that can accurately emulate various types of faults. Our fault injection experiment using this tool showed a reliability problem in software-based voters (i.e., used in N-modular redundancy). We thus have explored processor architecture extension techniques (*FDX*) to support various types of software-based hardware fault tolerance techniques. Two types of novel instructions were designed to increase the detection coverage and to accelerate error checking operations. The evaluation results showed that the presented system reduces the execution time of error detectors by 14% and their code size by 56%, and removes the identified TOCTTOU vulnerability windows. The *FDX* hardware area overhead is 0.7% compared with the baseline processor size. These high error detection and recovery coverage and low overheads in *FDX*-based systems are due to the co-design of hardware- and software-based fault tolerance techniques by using fault injection data.

Chapter 8.
Conclusion

This chapter concludes this dissertation.

8.1. Lessons Learned

This section summarizes the lessons learned from the fault characterization studies, the fault detection studies, and the overall design processes.

8.1.1. Fault Characterization

We have analyzed the characteristics of faults, errors, and failures. The characteristics are summarized as follows.

(i) Fault. Many faults are masked. When a target system is monitored for 30 seconds after a fault is injected, a large percentage (e.g., 72–99.7%) of the hardware faults that propagate to software-visible system architecture states in the processor and memory are benign and do not harm the reliability and data integrity of software. The large portion of masked errors may be due to the large system state spaces of software. In order to determine whether such masked faults are permanently masked or just not activated during the monitored time, we used a much longer monitoring time (e.g., 2.5 hours) and executed a stress test case suite of an OS. We observed lower ratios of masked faults, but most faults in the processor and memory (e.g., 72.9–97% of memory faults) were still masked. Such a high probability of fault masking in the architecture and software layers is due to: uncommonness of accessing certain portions of program states; probability of the first access after a corruption is write operation (i.e., error is removed by a new value); probability that the use of a corrupted data does not change the program behavior (e.g., checking $x > 0$ condition does not change the control flow as far as the corrupted value of x is larger than 0); and probability that a change in program behavior does not show up as an externally visible symptom (e.g., due to the absence of dataflow to the program output data).

(ii) Error. Baseline error detection techniques in modern computers are effective at detecting many errors induced by hardware faults. Our fault injection results show that a large portion of non-masked errors are detected (e.g., >88% of non-benign errors in a

Linux platform). For example, if a fault changes a call- or control-flow of software, the processor can fetch and decode an invalid instruction (e.g., stored in a data segment or due to misaligned access to a code segment). Such a situation is detected by the processor error detectors (i.e., invalid opcode exception). If either the instruction or the data is corrupted, this mismatch can produce a corrupted memory address operand, which is also detectable by the processor (e.g., segmentation fault or access permission violation). An error in an operand of an arithmetic instruction is detectable in some cases; for example, it is detectable if the denominator of a division instruction becomes zero. Furthermore, software-implemented error checkers in OSes detect errors that break the data integrity of kernel data structures and/or hardware states.

We observe different error sensitivity characteristics in GPU programs. Our fault injection experiments showed that an SEU emulated by injection of single-bit errors can seriously harm the reliability and data integrity of GPU kernels. The reason is that the percentage of data faults that can cause silent data corruption (SDC) errors in the evaluated GPU programs (e.g., 18–45%) is much larger than the percentage of data faults that can cause SDCs in common programs on CPUs.

(iii) Failure. It is possible to recover from a large portion of failures through use of simple software techniques. The reasons include short error latency, strong correlation between error and failure locations, and the significance of reproducible memory data. 95 to 97% of failures caused by processor faults have error latencies shorter than the time it takes to execute 5 billion instructions. 90 to 92.5% of such failures are detected in the same software module as the error occurrence. These failures are recoverable by a checkpoint-and-restart technique. However, 47% of failures caused by memory faults have fault latency longer than a time to execute 50 billion instructions. Such latent faults can cause a checkpoint corruption problem that needs an extra protection. Corruption in a portion of a checkpoint is tolerable by a forward error correction. For example, if a corrupted data was loaded from a disk and unmodified since then, this the problem is cured when the data are reloaded from the disk.

8.1.2. Fault Tolerance System Design

We have designed three types of fault tolerance frameworks. The key design principles are summarized as follows. Modern computer systems and software consist of techniques designed and implemented in multiple layers of system abstractions, so the fault tolerance techniques for these systems are as well. It is thus natural to use a hardware-software co-design approach and explore the wide hardware-software co-design spaces of fault tolerance techniques. The co-design approach in theory is likely to lead to a de-

sign that is closer to the global optimum than approaches that modify only a part of a system (e.g., software or hardware). We use the following design principles in order to explore the co-design space in an effective way.

(i) Co-design of error detection and recovery. Error detection and recovery algorithms are optimized at the same time. There are four cases in which this co-design provides benefits: error detection helps error recovery, error recovery helps error detection, an error recovery helps another error recovery, and an error detection helps another error detection. In a high layer of system abstraction, error isolation and recovery domains are defined and built using software and hardware techniques. The isolation property helps its error detection focus on the corruptions in the output data of each domain (namely, an *isolated execution and deferred checking* model). Furthermore, an error detection technique that is likely to limit the error detection latency helps its backward error recovery techniques optimize the recovery overhead (e.g., the number of checkpoints to keep).

The following examples are cases in which the error detection and recovery take advantage of each other. The examples show the benefits of our error detection and recovery co-design principle. An error isolation and recovery domain is a part of software and/or hardware that can isolate errors to its internal states and can tolerate the detected errors (e.g., by a re-execution). Some domain acts as a reliable computing base that can detect errors in other domains and trigger error recovery operations. A main technical challenge left in this fault tolerance architecture is that of detecting SDC errors in the output and input data of domains, because such data errors are undetected by the baseline fault tolerance techniques.

Let us assume that both forward and backward error recovery techniques are used. Then, a forward error recovery technique can selectively protect system states that are hard to recover from using the backward recovery technique, because forward error recovery (e.g., ECC) has overheads in normal fault-free conditions. By selecting protection targets based on knowledge of recoverability, that principle allows us to design a fault tolerance system that is closer to the global optimal, because the forward error recovery, backward error recovery, and error detection techniques are considered and optimized simultaneously.

(ii) Customized protection. We customize the error detection and recovery algorithms and techniques for each type of error isolation-recovery domain. For example, in this customization process, we use the error sensitivity, error recoverability, and protection cost of each type of target system state. This local customization principle was derived based on the observation that finding many local optima is easier than finding the global optimum. Here are some customization rules:

(a) Profiling-based customization. This sub-principle customizes the error detection and recovery algorithm by using profiling information on common programs of the targeted type of software or hardware in order to minimize the impact on system performance in common cases. For example, different error detection algorithms are used for different types of target software (e.g., parallel programs and privileged software). Such customization is also done at a finer granularity. Error detectors are strategically placed and customized in the source code of target GPU programs so as to minimize the performance impact and error propagation, and maximize recoverability. Specifically, in GPU programs, loops form a majority of program execution time (e.g., >90%). A lightweight error detection algorithm is thus designed for loop codes, while a powerful but expensive error detection algorithm is used for non-loop codes based on the underlying concepts in Amdahl's law [Amd67].

(b) Recoverability-driven selective protection. This sub-principle selectively protects part of system states. Example selection criteria are as follows. (1) A system state is selected for protection if the expected error detection and/or recovery coverage gain from protecting this state is higher than the coverage obtainable from protecting any of the remainder of unprotected states. For example, a data flow analysis is used to selectively protect program states to which errors in other states are likely to propagate. (2) We derive protection target states by considering the recoverability of errors in these states. For example, by considering the urgency in error detection to enable and support safe error recovery, an error detection technique can focus on protecting system states that are likely to cause long latency failures if a checkpointing-based error recovery technique is used.

(iii) Separation of algorithm and mechanism. In our co-design, we separate the fault tolerance algorithm and mechanism. Here, a fault tolerance *algorithm* means an effective method of detecting and tolerating errors, and a fault tolerance *mechanism* is the implementation of an algorithm in software and/or hardware (e.g., a finite sequence of instructions).

(a) Algorithm. An algorithm is heavily customized for a target subsystem and is strategically designed for targeted types of errors and failures. For example, an algorithm focuses on the detection of SDC errors. That allows us to reduce the coverage overlap with the baseline fault tolerance techniques of a target system (e.g., selectively targeting undetected or unrecoverable non-benign errors of a target system). Thus, many algorithms are designed and optimized in a system abstraction that is close to end users.

(b) Mechanism. A mechanism is optimized by exploiting the power of both hardware and software. For example, we offload fault tolerance operations to an external device that can efficiently process the operations without directly using the target system pro-

cessor cycles. In another example, we identify the most frequently used instruction subsequences of the initial software implementation of an algorithm and accelerate the identified sequences by using new processor instructions.

(c) Semantic gap. This separation of algorithm and mechanism allows designers to efficiently target non-benign errors without facing the semantic gap between hardware and software. Here, the semantic gap is the information gap between hardware and software. For example, although a hardware-implemented technique has the ability to access various system states, it is difficult to interpret the error sensitivity of each system state. By separating the algorithm and mechanism in different layers, the software-implemented algorithm can focus on identifying the non-benign errors and delivering this information to hardware-implemented mechanisms. The use of a standard hardware-software interface (e.g., an instruction set architecture extension in a case study) is effective at reducing the performance overhead associated with the information delivery because it can take advantage of existing hardware resources (e.g., memory hierarchy and computational parallelisms).

(iv) Transparency. We have designed fault tolerance techniques in a way that minimizes the deployment cost. For example, although the techniques are implemented as extensions of fundamental system components (i.e., processor, OS, and compiler), a user can directly take advantage of the techniques by compiling and launching the programs using provided tools.

8.1.3. Development Process

Based on those development studies, we present a novel development process (namely, a horseshoe model) that consists of the following eleven steps (see Figure 8.1):

1. ***User requirement analysis*.** This step is designed to analyze the user requirements on target system performance, cost, and energy efficiencies.
2. ***Tool development*.** This step develops a set of measurement tools, such as a fault injector and performance profiler, that can measure various types of target system characteristics.
3. ***Measurement*.** This step is develops measurement techniques that can help developers conduct spatially and temporally comprehensive measurement experiments by using the tools developed in the previous step. Here, *spatially comprehensive experiment* means an experiment that generalizes its findings on different types of computer systems. A temporally comprehensive experiment monitors the behaviors of a system a statistically significant number of times and/or for a sufficiently long period of time.

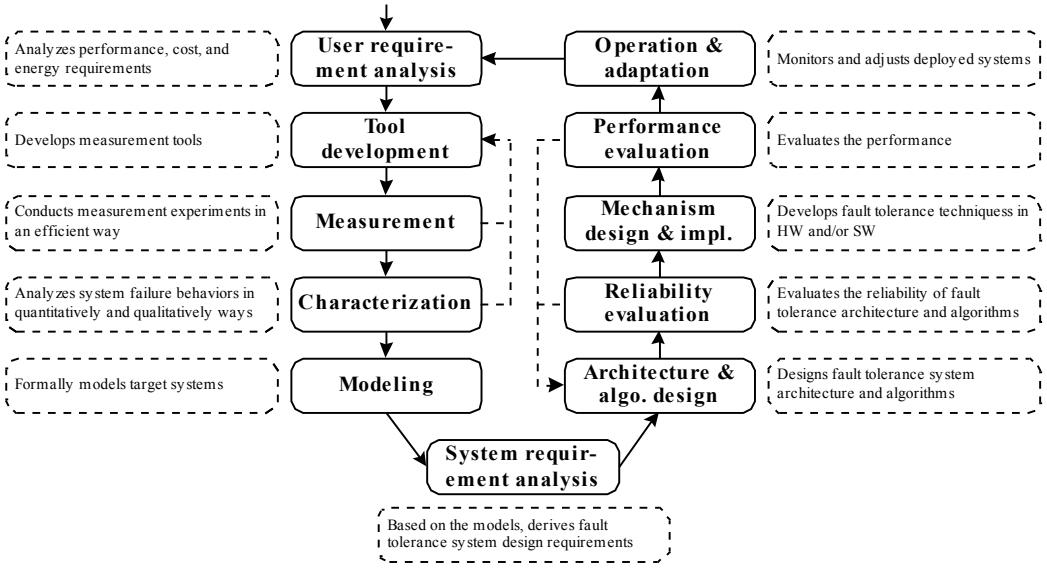


Figure 8.1. The presented development process.

4. ***Characterization.*** This step quantitatively and qualitatively analyzes the measured data. Quantitative analysis uses a set of metrics that can characterize the behaviors of system failures and normal executions. Qualitative analysis describes a set of cases that, for example, gives the propagation paths of an undetected error.
5. ***Modeling.*** This step develops a model by using the understanding (or expertise) obtained in the previous measurement-analysis steps and the quantized system characterization parameters. Considering the complexity of modern computer systems under various types of hardware faults, the model has a form of stochastic activity models and is solved by the Monte Carlo simulation method.
6. ***System requirement analysis.*** Using the developed model, this step explores the wide design space of fault tolerance techniques for a given configuration of target system hardware and software. This model-based design space exploration can derive the minimum design requirements of fault tolerance techniques that can meet the efficiency requirements derived by the user requirement analysis step.
7. ***Architecture and algorithm design.*** This step is designs the fault tolerance system architecture (e.g., boundaries between protection units) and fault tolerance approaches for each protection unit. For example, we decompose the system-level components and create error isolation and recovery domains in a way that reduces the execution time and the memory size of the single points of failure. One type of domains is for each GPU device, and the other type of domains is for the comput-

er node. In each domain, we optimize the algorithms and techniques of error detection and recovery by considering the characteristics of programs running on each domain.

8. ***Reliability evaluation.*** This step is evaluates the coverage of designed algorithms and their software implementation via fault injection experiments. The previous algorithm design step is revisited until the requirements are met.
9. ***Mechanism design and implementation.*** This step implements the designed algorithms in hardware and/or software. For example, in one of our case studies, we used the ASIP approach that identifies commonly used instruction sequences and accelerates them by adding a specialized computation engine (e.g., a coprocessor) to the target processor pipeline. Such hardware extension does not just accelerate execution of software-implemented fault tolerance algorithms, but also improves their coverage. In another case study, we offloaded fault tolerance techniques to a separate hardware device in the computer node, so that the fault tolerance techniques could run without directly using the computing power of target system.
10. ***Performance evaluation.*** This step evaluates the performance, memory, and energy overheads of the hardware-software co-designed fault tolerance techniques. The steps from 7 to 10 are repeated until all the derived system requirements are satisfied.
11. ***Operation.*** This step monitors the deployed fault tolerance systems in order to provide field evaluation data of the deployed techniques to administrators and/or users at runtime and to allow the administrators and users to dynamically select the deployed fault tolerance techniques and tune their configuration parameters.

8.2. Applications

We have presented various types of error detection and recovery techniques. In practice, it is possible to selectively use various combinations of these techniques to reflect the design requirements and configurations of a target system. The presented techniques can be grouped as functions of the types of protected subsystems.

- *CPU OS*. The software symptom and online clustering are a good combination to detect OS failures. One may use the existing heartbeat techniques instead of the online clustering. Because a failure of an OS leads to the failures of all its user processes, both multi-level CPU checkpointing and recoverability-driven memory protection are useful for tolerating errors and failures detected in an OS.

- *CPU MPI process*. All four types of the specification-based error detection techniques are useful for detecting errors and failures of MPI processes running on CPUs. If the use of all four checkers is constrained by the available resources, the use of the transition destination and time property checkers can be given top priority. In order to tolerate detected errors and failures of MPI processes, both the multi-level checkpointing and the recoverability-driven memory protection are useful.
- *GPU thread*. The GPU guardian technique is useful for detecting hang failures and long execution errors of GPU threads. In order to provide stronger protection for SDC errors, one can use the similarity-based technique if a target program has a good spatial or temporal value similarity property (e.g., *n*-body program). For target programs that do not have such a property, the embedded error-checking techniques can be used as an alternative that can provide strong error detection coverage. If hardware modification is possible, one can also take advantage of the hardware-software co-designed techniques and run the embedded error checkers in a more efficient way. The selective re-execution technique and a GPU checkpoint technique are a good combination to offer good recovery from errors and failures in GPUs.
- *Mission-critical systems*. In mission-critical systems that use a relatively small number of computer nodes, the presented software-based NMR alone is good enough to provide strong error detection and recovery coverage.

Depending on the scale of target system, there are three possible ways to combine the presented techniques. The cost-efficient fault tolerance techniques (i.e., all presented techniques except for the NMR) can improve the system efficiencies until a target parallel or distributed system reaches a certain scale. When the system scale exceeds that threshold, hardware failures become so common that they can even block the executions of many of the presented techniques (e.g., checkpoint-restart operation). In such a case, the presented software-based NMR technique can be used in lower layers of system abstraction to detect and tolerate many hardware faults. Note that there may be alternative approaches (e.g., failure-oblivious computing [RCD+04]) that can tolerate such a high rate of faults with computation and energy efficiencies higher than 33.3%. That possibility is not extensively explored in this dissertation because of our focus on transparency. For example, we want to be able to run legacy MPI programs in new larger-scale computers without direct modifications to the source code or programming model. As the system scale grows, despite the use of the NMR technique, failures can become so common that they cannot be tolerated by the baseline techniques (e.g., heartbeat and checkpoint) in

an efficient way. In that third case, both the cost-efficient fault tolerance techniques and the NMR technique need to be deployed at the same time. A combination of those two types of techniques thus can significantly shift the scaling limit of parallel or distributed computer systems.

Some of the presented frameworks and techniques can be used to detect failures caused by software faults and security attacks. For example, the presented hang detection techniques can detect failures of software that are caused by denial of service (DoS) and distributed DoS attacks. The presented SDC detection techniques can detect malicious intruders who harm the data integrity of software at runtime (e.g., memory corruption). In this dissertation, we limited our target system to parallel computer systems. Thus, there remains an important open challenge of finding a way to apply the presented experimental validation and design methodologies to large-scale distributed computer systems (e.g., MapReduce) that have similar scalability, reliability, and availability problems.

References

[ACR+08] P. Akritidis, C. Cadar, C. Raiciu, et al., "Preventing Memory Error Exploits with WIT," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2008.

[AFR02] J. Arlat, J.-C. Fabre, and M. Rodriguez, "Dependability of COTS Microkernel-Based Systems," *IEEE Transactions on Computers*, 51(2):138-163, 2002.

[ALL06] K. Asrigo, L. Litty, and D. Lie, "Using VMM-Based Sensors to Monitor Honeypots," in *Proceedings of the International Conference on Virtual Execution Environments*, pp. 13-23, 2006.

[ALR+04] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic Concepts and Taxonomy of Dependable and Secure Computing," *IEEE Transactions on Dependable and Secure Computing*, 1(1):11-33, 2004.

[AMA+11] F. Azmandian, M. Moffie, M. Alshawabkeh, J. Dy, J. Aslam, and D. Kaeli, "Virtual machine monitor-based lightweight intrusion detection," *ACM SIGOPS Operating Systems Review*, 45(2):38-53, 2011.

[Amd67] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the AFIPS Spring Joint Computer Conference*, 1967.

[AKT+08] H. Ando, R. Kan, Y. Tosaka, K. Takahisa, and K. Hatanaka, "Validation of Hardware Error Recovery Mechanisms for the SPARC64 V Microprocessor," in *Proceedings of the International Conference on Dependable Systems and Networks*, pp. 62-69, 2008.

[Aus99] T. M. Austin, "DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design," in *Proceedings of the International Symposium on Microarchitecture*, pp. 196-207, 1999.

[Avi85] A. Avizienis, "The N-Version Approach to Fault-Tolerant Software," *IEEE Transactions on Software Engineering*, 11(12):1491-1501, 1985.

[Bag01] S. Bagchi, *Hierarchical Error Detection in a Software Implemented Fault Tolerance (SIFT) Environment*, Ph.D. Dissertation, University of Illinois at Urbana-Champaign, 2001.

[BC00] D. P. Bovet and M. Cesati, *Understanding the Linux Kernel*, 3rd Ed., O'Reilly Media, 2000.

[BCS+90] J. H. Barton, E. W. Czeck, Z. Z. Segall, and D. P. Siewiorek, "Fault Injection Experiments Using FIAT," *IEEE Transactions on Computers*, 39(4):575-582, 1990.

[BGG+02] J. C. Baraza, J. Gracia, D. Gil, and P. J. Gil, "A Prototype of a VHDL-based Fault Injection Tool: Description and Application," *Journal of Systems Architecture*, 47(10):847-867, 2002.

[BL94] V. Barnett and T. Lewis, *Outliers in Statistical Data*, John Wiley, 1994.

[BSS08] L. Borucki, G. Schindlbeck, and C. Slayman, "Comparison of Accelerated DRAM Soft Error Rates Measured at Component and System Level," in *Proceeding of the International Reliability Physics Symposium*, pp. 482-487, 2008.

[But83] R. Butler, "Outlier Discordancy Test in the Normal Linear Model," *Journal of the Royal Statistical Society*, B-45(1):120-132, 1883.

[CB89] R. Chillarege and N. S. Bowen, "Understanding Large System Failures - A Fault Injection Experiment," in *Proceedings of the International Symposium on Fault-Tolerant Computing*, pp. 356-363, 1989.

[CCM+09] M. Castro, M. Costa, J.-P. Martin, et al., "Fast Byte-Granularity Software Fault Isolation," in *Proceedings of the ACM Symposium on Operating Systems Principles*, pp. 45-58, 2009.

[CFH+04] J. Cong, Y. Fan, G. Han, and Z. Zhang, "Application Specific Instruction Generation for Configurable Processor Architectures," in *Proceedings of the International Symposium on Field-Programmable Gate Arrays*, pp. 183-189, 2004.

[CI87] R. Chillarege and R.K. Iyer, "Measurement-Based Analysis of Error Latency," *IEEE Transactions on Computers*, C-36(5):529-537, 1987.

[CI92] G. Choi and R. K. Iyer, "FOCUS: An Experimental Environment for Fault Sensitivity Analysis," *IEEE Transactions on Computers*, 41(12):1515-1526, 1992.

[CKV+04] G. Chen, M. Kandemir, N. Vijaykrishnan, A. Sivasubramaniam, and M. J. Irwin, "Analyzing Heap Error Behavior in Embedded JVM Environments," in *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis*, pp. 230-235, 2004.

[CMS98] J. Carreira, H. Madeira, and J. G. Silva, "Xception: Software Fault Injection and Monitoring in Processor Functional Units," *IEEE Transactions on Software Engineering*, 24(2):125-136, 1998.

[Con03] C. Constantinescu, “Trends and challenges in VLSI circuit reliability,” *IEEE Micro*, 23(4):14–19, 2003.

[CR07] F. Chen and G. Rosu, “MOP: An efficient and generic runtime verification framework,” in *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages and Applications*, pp. 569-588, 2007.

[CR11] G. M. Castillo and B. A. Ratkevich, “Single Event Upset Testing of Commercial Off-The-Shelf Electronics for Launch Vehicle Applications,” in *Proceedings of the IEEE Aerospace Conference*, 2011

[CRA06] J. Chang, G. A. Reis, and D. I. August, “Automatic Instruction-Level Software-Only Recovery Methods,” in *Proceedings of the International Conference on Dependable Systems and Networks*, pp. 83-92, 2006.

[DLP03] J. J. Dongarra, P. Luszczek, and A. Petitet, "The LINPACK Benchmark: Past, Present, and Future," *Concurrency and Computation: Practice and Experience*, ISSN 1532-0634, 15(9):803-820, 2003.

[DM06] J. A. Durães and H. S. Madeira, “Emulation of Software Faults: A Field Data Study and a Practical Approach,” *IEEE Transactions on Software Engineering*, 32(11):849-867, 2006.

[DMZ09] M. Dimitrov, M. Mantor, and H. Zhou, “Understanding Software Approaches for GPGPU Reliability,” in *Proceedings of the Workshop on General Purpose Processing on Graphics Processing Units*, pp. 94-104, 2009.

[ECG+01] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, “Dynamically Discovering Likely Program Invariants to Support Program Evolution,” *IEEE Transactions on Software Engineering*, 27(2):99-123, 2001.

[EKD+03] D. Ernst, N. S. Kim, S. Das, S. Pant, R. Rao, T. Pham, C. Ziesler, D. Blaauw, T. Austin, K. Flautner, and T. Mudge, “Razor: A Low-Power Pipeline Based on Circuit-Level Timing Speculation,” in *Proceedings of the International Symposium on Microarchitecture*, 2003.

[FGA+10] S. Feng, S. Gupta, A. Ansari, and S. Mahlke, “Shoestring: probabilistic soft error reliability on the cheap,” in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 385-396, 2010.

[FHS+96] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff, “A Sense of Self for Unix Processes,” in *Proceedings of the IEEE Symposium on Security and Privacy*, pp. 120-128, 1996.

[FPP78] D. Freedman, R. Pisani, and R. Purves, *Statistics*, W. W. Norton, New York, 1978.

[GCG+07] J. N. Glosli, K. J. Caspersen, J. A. Gunnels, D. F. Richards, R. E. Rudd, and F. H. Streitz, "Extending Stability Beyond CPU Millennium: A Micron-Scale Atomistic Simulation of Kelvin-Helmholtz Instability," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, pp. 1-11, 2007.

[GGR04] M. S. Gordon, P. Goldhagen, K. P. Rodbell, T. H. Zabel, H. H. K. Tang, J. M. Clem, and P. Bailey, "Measurement of the flux and energy spectrum of cosmic-ray induced neutrons on the ground," *IEEE Transactions on Nuclear Science*, 51(6):3427-3434, 2004.

[GIY97] K. K. Goswami, R. K. Iyer, and L. Young, "DEPEND: A Simulation-Based Environment for System-Level Dependability Analysis," *IEEE Transactions on Computers*, 46(1):60-74, 1997.

[GJ95] A. K. Ghosh and B. W. Johnson, "System-Level Modeling in the ADEPT Environment of a Distributed Computer System for Real-time Applications," in *Proceedings of the IEEE International Computer Performance and Dependability Symposium*, pp. 194-203, 1995.

[GKI04] W. Gu, Z. Kalbarczyk, and R.K. Iyer, "Error Sensitivity of the Linux Kernel Executing on PowerPC G4 and Pentium 4 Processors," in *Proceedings of the International Conference on Dependable Systems and Networks*, pp. 887-896, 2004.

[GSV05] R. Gopalakrishna, E. H. Spafford, and J. Vitek, "Efficient Intrusion Detection Using Automaton Inlining," in *Proceedings of the IEEE Symposium on Security and Privacy*, pp. 18-31, 2005.

[HA84] K. Huang, J. Abraham, "Algorithm-Based Fault Tolerance for Matrix Operations," *IEEE Transactions on Computers*, C-33(6):518-528, 1984.

[HJS02] M. Hiller, A. Jhumka, and N. Suri, "On the Placement of Software Mechanisms for Detection of Data Errors," in *Proceedings of the International Conference on Dependable Systems and Networks*, pp. 135-144, 2002.

[HP10] I. S. Haque and V. S. Pande, "Hard Data on Soft Errors: A Large-Scale Assessment of Real-World Error Rates in GPGPU," in *Proceedings of the IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, pp. 691-696, 2010.

[IPMI2] IPMI Alliance (Intel, Hewlett-Packard, NEC, and Dell), "IPMI – Intelligent Platform Management Interface Specification Second Generation v2.0," IPMI Specification Document, 2004.

[IT96] R. K. Iyer and D. Tang, "Experimental Analysis of Computer Sys-tem Dependability," *Fault-Tolerant Computer System Design*, Ch. 5, D.K. Pradhan (ed.), Prentice Hall, 1996.

[JAZ+10] M. Jovic, A. Adamoli, D. Zaparanuks, and M. Hauswirth, "Automating Performance Testing of Interactive Java Applications," in *Proceedings of the International Workshop on Automation of Software Test*, pp. 8-15, 2010.

[JIH+97] H. Jin, R.K. Iyer, M.C. Hsueh, and M. Covington, "FAMAS: Fault Modeling via Adaptive Simulation," in *Proceedings of the International Conference on VLSI Design*, pp. 413-441, 1997.

[KA95] G. Kanawati and J. A. Abraham, "FERRARI: A Flexible Software-based Fault and Error Injection System," *IEEE Transactions on Computers*, 44(2):248-260, 1995.

[KIR+99] Z. Kalbarczyk, R. K. Iyer, G. L. Ries, J. U. Patel, M. S. Lee, and Y. Xiao, "Hierarchical Simulation Approach to Accurate Fault Modeling for System Dependability Evaluation," *IEEE Transactions on Software Engineering*, 25(5):619-632, 1999.

[KIT93] W.-l. Kao, R. K. Iyer, and D. Tang, "FINE: A Fault Injection and Monitoring Environment for Tracing the UNIX System Behavior under Faults," *IEEE Transactions on Software Engineering*, 19(11):1105-1118, 1993.

[KKA95] G. A. Kanawati, N. A. Kanawati, and J. A. Abraham, "FERRARI: A Flexible Software-Based Fault and Error Injection System," *IEEE Transactions on Computers*, 44(2):248-260, 1995.

[KN98] E. M. Knorr and R. T. Ng, "Algorithms for Mining Distance-Based Outliers in Large Datasets," in *Proceedings of the International Conference on Very Large Data Bases*, pp. 392-403, 1998.

[KNR09] E. H. Kim, J. C. Na, and S. M. Ryoo, "Implementing an Effective Test Automation Framework," in *Proceedings of the IEEE International Computer Software and Applications Conference*, pp. 534-538, 2009.

[KNS10] M. de Kruijf, S. Nomura, and K. Sankaralingam, "Relax: An Architectural Framework for Software Recovery of Hardware Faults," in *Proceedings of the International Symposium on Computer Architecture*, pp. 497-508, 2010.

[KS01] M. D. Krstic and M. K. Stojcev, "FPGA Implementation of Hardware Voter," in *Proceedings of International Conference on Telecommunications in Modern Satellite, Cable and Broadcasting Services*, pp. 401-404, 2001.

[Lap95] J.-C. Laprie, "Dependable Computing: Concepts, Limits, Challenges," in *Proceedings of the IEEE International Symposium on Fault-Tolerant Computing*, Special Issue, pp. 42-54, 1995.

[LR04] C.-d. Lu and D. A. Reed, "Assessing Fault Sensitivity in MPI Applications," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, No. 37, 2004.

[LRS+08] M.-L. Li, P. Ramachandran, S. K. Sahoo, S. Adve, V. Adve, and Y. Zhou, "Understanding the Propagation of Hard Errors to Software and Implications for Resilient System Design," in *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 265-276, 2008.

[LSC97] W. Lee, S. J. Stolfo, and P. K. Chan, "Learning Patterns from Unix Process Execution Traces for Intrusion Detection," in *Proceedings of AAAI Workshop: AI Approaches to Fraud Detection and Risk Management*, 1997.

[LSH+07] X. Li, K. Shen, M. Huang, and L. A. Chu, "Memory soft error measurement on production systems," in *Proceedings of USENIX Annual Technical Conference*, No. 21, 2007.

[LSM99] W. Lee, S. J. Stolfo, and K. W. Mok, "A Data Mining Framework for Building Intrusion Detection Models," in *Proceedings of the IEEE Symposium on Security and Privacy*, pp. 120-132, 1999.

[Mes82] G. C. Messenger, "Collection of Charge on Junction Nodes from Ion Tracks," *IEEE Transactions on Nuclear Science*, 29(6):2024-2031, 1982.

[MHH+05] 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 Laboratory's ASC Q Computer," *IEEE Transactions on Device and Materials Reliability*, 5(3):329-335, 2005.

[MM00] S. Mitra and E. J. McCluskey, “WORD-VOTER: A New Voter Design for Triple Modular Redundant Systems,” in *Proceedings of the VLSI Test Symposium*, pp. 465-470, 2000.

[MM10] N. Miskov-Zivanov and D. Marculescu, “Multiple Transient Faults in Combinational and Sequential Circuits: A System Approach,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 29(10):1614-1627, 2010.

[MM88] A. Mahmood and E. J. McCluskey, “Concurrent error detection using watchdog processors-a survey,” *IEEE Transactions on Computers*, 37(2):160-174, 1988.

[MNM10] N. Maruyama, A. Nukada, and S. Matsuoka, “A high-performance fault-tolerant software framework for memory on commodity GPUs,” in *Proceedings of the International Symposium on Parallel and Distributed Processing*, pp. 1-12, 2010.

[MRM+94] H. Madeira, M. Rela, F. Moreira, and J. G. Silva, “RIFLE: A General Purpose Pin-Level Fault Injector,” *Lecture Notes in Computer Science*, 852:199-216, 1994.

[MWE+03] 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 International Symposium on Microarchitecture*, pp. 29-40, 2003.

[NKI+04] N. Nakka, Z. Kalbarczyk, R. K. Iyer, and J. Xu, “An Architectural Framework for Providing Reliability and Security Support,” in *Proceedings of the International Conference on Dependable Systems and Networks*, pp. 585-594, 2004.

[NPB08] D. Nowroth, I. Polian, and B. Becker, “A Study of Cognitive Resilience in a JPEG Compressor,” in *Proceedings of the International Conference on Dependable Systems and Networks*, pp. 32-41, 2008.

[ORT+96] T. J. O'Gorman, J. M. Ross, A. H. Taber, J. F. Ziegler, H. P. Muhlfeld, C. J. Montrose, M. W. Curtis, and J. L. Walsh, “Field testing for cosmic ray soft errors in semiconductor memories,” *IBM Journal of Research and Development*, 40(1):41-50, 1996.

[OSM02] N. Oh, P. P. Shirvani, and E. J. McCluskey, “Error Detection by Duplicated Instructions in Super-Scalar Processors,” *IEEE Transactions on Reliability*, 51(1)63-75, 2002.

[PKI07] K. Pattabiraman, Z. Kalbarczyk, and R. K. Iyer, "Automated Derivation of Application-aware Error Detectors using Static Analysis," in *Proceedings of the International On-Line Testing Symposium*, pp. 211-216, 2007.

[RBC+92] R. Chillarege, I. S. Bhandari, J. K. Chaar, M. J. Halliday, D. S. Moebus, B. K. Ray, M.-Y. Wong, "Orthogonal Defect Classification – A Concept for In-Progress Measurements," *IEEE Transactions on Software Engineering*, 18(11):943-956, 1992.

[RCD+04] M. Rinard, C. Cadar, D. Dumitran, D. M. Roy, T. Leu, and W. S. Beebee Jr., "Enhancing Server Availabilty and Security Through Failure-Oblivious Computing," in *Proceedings of the USENIX Symposium on Operating System Design and Implementation*, No. 21, 2004.

[RCV+05] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August, "SWIFT: Software Implemented Fault Tolerance," in *Proceedings of the International Symposium on Code Generation and Optimization*, pp. 243-254, 2005.

[RPK05] R. G. Ragel, S. Parameswaran, and S. M. Kia, "Micro Embedded Monitoring for Security in Application Specific Instruction-set Processors," in *Proceedings of the International Conference on Compilers, Architectures, and Synthesis of Embedded Systems*, pp. 304-314, 2005.

[RR96] I. Ruts and P. Rousseeuw, "Computing Depth Contours of Bivariate Point Clouds," *Journal of Computational Statistics and Data Analysis*, 23:153-168, 1996.

[RRK00] S. Ramaswamy, R. Rastogi, and S. Kyuseok, "Efficient Algorithms for Mining Outliers from Large Data Sets," in *Proceedings of ACM SIGMOD International Conference on Management of Data*, pp. 427-438, 2000.

[SAC+99] 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," *IEEE Micro*, 19(2):12-23, 1999.

[SFA+07] G. F. Sacco, R. D. Ferraro, P. v. Allmen, and D. A. Rennels, "Fault Injection Campaign for a Fault Tolerant Duplex Framework," in *Proceedings of the Aerospace Conference*, 2007.

[SES+09] G. Shi, J. Enos, M. Showerman, and V. Kindratenko, "On Testing GPU Memory for Hard and Soft Errors," in *Proceedings of the Symposium on Application Accelerators in High-Performance Computing*, 2009.

[SFB+00] D. T. Stott, B. Floering, D. Burke, Z. Kalbarczyk, and R. K. Iyer, "NFTAPE: A Framework for Assessing Dependability in Distributed Systems with Lightweight Fault Injectors," in *Proceedings of International Computer Performance and Dependability Symposium*, pp. 91-100, 2000.

[SG06] B. Schroeder and G. A. Gibson, "A Large Scale Study of Failures in High-Performance Computing Systems," in *Proceedings of the International Conference on Dependable Systems and Networks*, pp. 249-258, 2006.

[SG07] B. Schroeder and G. A. Gibson, "Disk failures in the real world: What does an MTTF of 1,000,000 hours mean to you?," in *Proceedings of the USENIX Conference on File and Storage Technologies*, No. 1, 2007.

[SKB12] J. Sloan, R. Kumar, and G. Bronevetsky, "Algorithmic Approaches to Low Overhead Fault Detection for Sparse Linear Algebra," in *Proceedings of the International Conference on Dependable Systems and Networks*, 2012.

[SL86] K. G. Shin and Y. H. Lee, "Measurement and Application of Fault Latency," *IEEE Transactions on Computers*, C-35(4):370-375, 1986.

[SLR+08] S. K. Sahoo, M.-L. Li, P. Ramachandran, S. Adve, V. Adve, and Y. Zhou, "Using Likely Program Invariants to Detect Hardware Errors," in *Proceedings of the International Conference on Dependable Systems and Networks*, pp. 70-79, 2008.

[SLS06] J. W. Sheaffer, D. P. Luebke, and K. Skadron, "The Visual Vulnerability Spectrum: Characterizing Architectural Vulnerability for Graphics Hardware," in *Proceedings of the ACM SIGGRAPH Symposium on Graphics Hardware*, pp. 9-16, 2006.

[SLS07] J. W. Sheaffer, D. P. Luebke, and K. Skadron, "A Hardware Redundancy and Recovery Mechanism for Reliable Scientific Computation on Graphics Processors," in *Proceedings of the ACM SIGGRAPH Symposium on Graphics Hardware*, pp. 55-64, 2007.

[SMR+07] A. Shye, T. Moseley, V. J. Reddi, J. Blomstedt, and D. A. Connors, "Using Process-Level Redundancy to Exploit Multiple Cores for Transient Fault Tolerance," in *Proceedings of the International Conference on Dependable Systems and Networks*, pp. 297-306, 2007.

[SPP+04] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, D. Boneh, "On the Effectiveness of Address-Space Randomization," in *Proceedings of the ACM Conference on Computer and Communications Security*, pp. 298-307, 2004.

[SPW09] B. Schroeder, E. Pinheiro, and W.-D. Weber, "DRAM Errors in the Wild: A Large-Scale Field Study," in *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*, pp. 193-204, 2009.

[SSC03] T. Sherwood, S. Sair, and B. Calder, "Phase Tracking and Prediction," in *Proceedings of the International Symposium on Computer Architecture*, pp. 336-347, 2003.

[SSP90] A. M. Saleh, J. J. Serrano, and J. H. Patel, "Reliability of Scrubbing Recovery-Technique for Memory Systems," *IEEE Transactions on Reliability*, 39(1):114-122, 1990.

[SVK+05] G. P. Saggese, A. Vetteth, Z. Kalbarczyk, and R. Iyer, "Microprocessor Sensitivity to Failures: Control vs Execution and Combinational vs Sequential Logic," in *Proceedings of International Conference on Dependable Systems and Networks*, pp. 760-769, 2005.

[TEF+05] D. Tsafrir, Y. Etsion, D. G. Feitelson, and S. Kirkpatrick, "System Noise, OS Clock Ticks, and Fine-grained Parallel Applications," in *Proceedings of the International Conference on Supercomputing*, pp. 303-312, 2005.

[Tri01] K. S. Trivedi, *Probability and Statistics with Reliability, Queuing, and Computer Science Applications*, Wiley-Interscience, 2001.

[Tuk77] J. W. Tukey, Exploratory Data Analysis, Addison-Wesley, 1977.

[WD01] D. Wagner and D. Dean, "Intrusion Detection via Static Analysis," in *Proceedings of the IEEE Symposium on Security and Privacy*, pp. 156-168, 2001.

[WP05] N. Wang and S. J. Patel, "ReStore: A Symptom-Based Processor Architecture for Soft-Error Tolerance," in *Proceedings of the International Symposium on Dependable Systems and Networks*, 2005.

[WQR+04] N. J. Wang, J. Quek, T. M. Rafacz, and S. J. Patel, "Characterizing the Effects of Transient Faults on a High-Performance Processor Pipeline," in *Proceedings of the International Conference on Dependable Systems and Networks*, pp. 61-70, 2004.

[Yan92] F. Yang, *Simulation of Faults Causing Analog Behavior in Digital Circuits*, PhD Dissertation, University of Illinois at Urbana-Champaign, 1992.

[YKI09] K. S. Yim, Z. T. Kalbarczyk, and R. K. Iyer, "Quantitative Analysis of Long Latency Failures in System Software," in *Proceedings of the Pacific-Rim International Symposium on Dependable Computing*, pp. 23-30, 2009.

[YKI10] K. S. Yim, Z. Kalbarczyk, and R. K. Iyer, "Measurement-based Analysis of Fault and Error Sensitivities of Dynamic Memory," in *Proceedings of the International Conference on Dependable Systems and Networks*, pp. 431-436, 2010.

[YPS+11] K. S. Yim, C. Pham, M. Saleheen, Z. Kalbarczyk, and R. Iyer, "Hauberk: Lightweight Silent Data Corruption Error Detector for GPGPU," in *Proceedings of the International Parallel and Distributed Processing Symposium*, pp. 287-300, 2011.

[YSD+09] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar, "Native Client: A Sandbox for Portable, Untrusted x86 Native Code," in *Proceedings of the IEEE Symposium on Security and Privacy*, pp. 79-93, 2009.

[YZG00] J. Yang, Y. Zhang, and R. Gupta, "Frequent Value Compression in Data Caches," in *Proceedings of the International Symposium on Microarchitecture*, pp. 258-265, 2000.

[ZCM+96] 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*, 40(1):3-18, 1996.

[Zie96] J. F. Ziegler, "Terrestrial cosmic rays," *IBM Journal of Research and Development*, 40(1):19-39, 1996.

[ZMM+96] J. F. Ziegler, H. P. Muhlfeld, C. J. Montrose, H. W. Curtis, T. J. O'Gorman, and J. M. Ross, "Accelerated testing for cosmic soft-error rate," *IBM Journal of Research and Development*, 40(1):51-72, 1996.

[ZNS+98] J. F. Ziegler, M. E. Nelson, J. D. Shell, R. J. Peterson, C. J. Gelderloos, H. P. Muhlfeld, and C. J. Montrose, "Cosmic ray soft error rates of 16-Mb DRAM memory chips," *IEEE Journal of Solid-State Circuits*, 33(2):246-252, 1998.