2001

Affordable techniques for dependable microprocessor design

Seongwoo Kim
Iowa State University

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

Affordable Techniques for Dependable Microprocessor Design

by

Seongwoo Kim

A dissertation submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Major: Computer Engineering

Program of Study Committee:
Arun K. Somani, Major Professor
Suraj C. Kothari
Gyungho Lee
Jo Min
Akhilesh Tyagi

Iowa State University

Ames, Iowa

2001

UMI Number: 3034195

ProQuest Information and Learning Company
300 North Zeeb Road
P.O. Box 1346
Ann Arbor, MI 48106-1346

Graduate College
Iowa State University

This is to certify that the Doctoral dissertation of

Seongwoo Kim

has met the dissertation requirements of Iowa State University

Signature was redacted for privacy.

Major Professor

Signature was redacted for privacy.

For the Major Program

DEDICATION

In memory of my father, Woonki

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I express my deep gratitude to my advisor, Professor Arun K. Somani, for his tremendous guidance, support, and encouragement throughout my graduate study at Iowa State and Washington. His professional advice and insights always led me to the right directions in research and career. He has always been my role model.

I would like to sincerely thank Prof. Shin-dug Kim for introducing me to the most fascinating world of computer engineering. I wish to thank Prof. Oliver Diessel, Prof. Soohwan Jung, and Dr. Donglok Kim for his encouragement and invaluable advice. I also thank my colleagues at the Dependable Computing and Networking Laboratory (DCNL) for constructive discussions and companionship through the years: Ling Li, Stefano Chessa, Govindarajan Krishnamurthy, Sashisekaran Thiagarajan, Murari Sridharan, Srinivasan Ramasubramanian, Liang Zhao, Joel Nickel, Rama Sangireddy, Anirban Chakrabarti, Aaron Striegel, Jing Fang, Huesung Kim, and Wu Tao. Special thanks to Matt Virgo, Jason Thomas, Aaron Cordes, Jon E. Froehlich, and Mei-Peng Cheong for being our undergraduate research assistants, working with me on the research projects, and sharing their enthusiasm for active learning.

Finally, I am very grateful to Siwon, my wife, for her love and constant support. I cannot thank her enough for successfully managing our Ames life without family, old friends, and joyful life in Seoul. I also thank my mother for being such a super-mom, and Younghee Noh, Kwangyeon Shin, Nanyoung, and Youngeun for their love and support from Seoul.

ABSTRACT

As high computing power is available at an affordable cost, we rely on microprocessor-based systems for much greater variety of applications. This dependence indicates that a processor failure could have more diverse impacts on our daily lives. Therefore, dependability is becoming an increasingly important quality measure of microprocessors.

Temporary hardware malfunctions caused by unstable environmental conditions can lead the processor to an incorrect state. This is referred to as a transient error or soft error. Studies have shown that soft errors are the major source of system failures. This dissertation characterizes the soft error behavior on microprocessors and presents new microarchitectural approaches that can realize high dependability with low overhead.

Our fault injection studies using RISC processors have demonstrated that different functional blocks of the processor have distinct susceptibilities to soft errors. The error susceptibility information must be reflected in devising fault tolerance schemes for cost-sensitive applications. Considering the common use of on-chip caches in modern processors, we investigated area-efficient protection schemes for memory arrays. The idea of caching redundant information was exploited to optimize resource utilization for increased dependability. We also developed a mechanism to verify the integrity of data transfer from lower level memories to the primary caches. The results of this study show that by exploiting bus idle cycles and the information redundancy, an almost complete check for the initial memory data transfer is possible without incurring a performance penalty.

For protecting the processor's control logic, which usually remains unprotected, we propose a low-cost reliability enhancement strategy. We classified control logic signals into static and dynamic control depending on their changeability, and applied various techniques including

commit-time checking, signature caching, component-level duplication, and control flow monitoring. Our schemes can achieve more than 99% coverage with a very small hardware addition. Finally, a virtual duplex architecture for superscalar processors is presented. In this system-level approach, the processor pipeline is backed up by a partially replicated pipeline. The replication-based checker minimizes the design and verification overheads. For a large-scale superscalar processor, the proposed architecture can bring 61.4% reduction in die area while sustaining the maximum performance.

CHAPTER 1 Introduction

1.1 Background

Microprocessors are used in a wide variety of applications from small calculators to multi-million dollar servers. As we become more dependent upon microprocessor-based systems, increasing attention is paid not only to the processor's computing speed but also to its dependability. *Dependability* is the trait of being dependable including reliability, availability, serviceability, safety, security, trustability, and many other capabilities that the users expect from the system. Among those, reliability and availability are most frequently used dependability measures. Reliability is defined as the probability of continuous operation over an interval [0,T], whereas availability is the proportion of time the system is available to perform useful work.

Modern electronic circuits are extremely complex and, as such, are susceptible to errors and failures. Even if a microprocessor is shipped with no design errors or manufacturing defects, unstable environmental conditions can generate temporary hardware failures. These failures, called *transient faults*, can cause the processor to malfunction during operation time. Traditionally, permanent hardware failures have been greater concern of the processor designers and users, but several studies have revealed that the vast majority of detected errors originate from the transient faults [1], [2], [3]. The important sources of the transient faults are electromagnetic interference, power jitter, and decay of radioactive atoms such as alpha particles and extraterrestrial cosmic rays constantly bombarding the earth [4], [5]. Although the transient faults are becoming a primary cause of system failure, they are not perceived as often as they affect the system by end users. This is because the users generally observe erroneous behavior of the system only through the software interface. For example, we simply blame the operating system for process hanging on our PCs. However, transient hardware faults can also be the source of the failure. The publications on the actual transient failure rates in commercial microprocessors are seldom found. The manufacturers usually consider them as proprietary

knowledge.

Although no comprehensive database exists for the dependability investigation of commercial products, we can easily infer serious soft error problems from physical nature of microelectronic circuits. Advances in VLSI technology have shrunk circuit dimensions and improved the processor performance dramatically, yet these advances are offset by an increased vulnerability to soft errors [6], [7]. This is more apparent from a basic relation of $Q_{crit} = C \times V$, where critical charge Q_{crit} of a digital circuit element is the minimum charge needed to change the element's logic state; since both capacitance (C) and voltage (V) reduce with future technologies, Q_{crit} also decreases, and thus, the probability that random noises disrupt the circuit significantly increases. In addition, increasing design complexity, doubling transistor density, low energy per signal transition, and extremely fast clock cycles magnify the processor's soft error sensitivity. Figure 1.1 illustrates that each new technology scaling almost doubles soft error rate in an SRAM. A similar increase is expected for the processors.

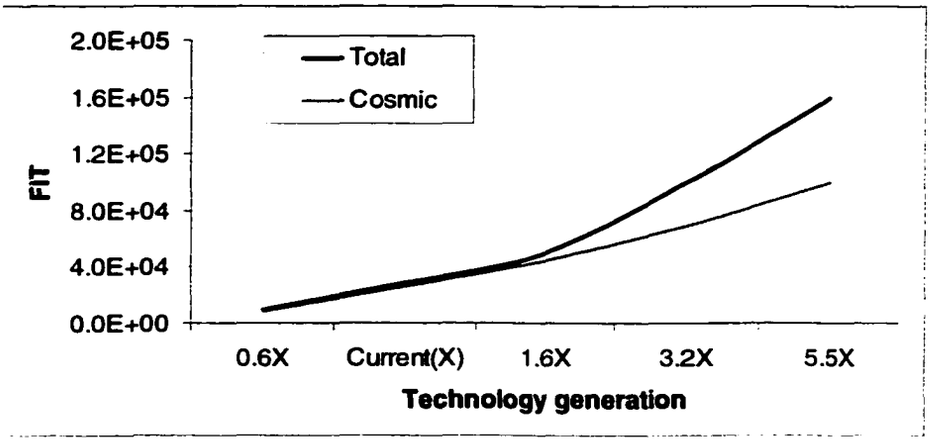


Figure 1.1 FIT (failures in 10^9hours) increase of a SRAM cache in scaling IC process technology generation.

An incorrect state of a processor component manifesting transient faults is referred to as a *soft error*. Soft errors in complex microprocessors have various effects. Although every element may be susceptible to the soft error, it should be noted that not all errors result in failures. For example, a corrupted register value by a transient fault can be overwritten before it is used. Errors may be masked without any intended efforts. Nevertheless, the fact that even a single-bit error may cause fatal damage creates the need of error detection and recovery capabilities. Soft errors need to be continuously monitored throughout the chips' lifetime. Keeping the chip dependability under control is critical and also challenging to fully appreciate the benefits of

deep submicron scaling. Unfortunately, research in the area of the processor dependability has been generally limited to a few critical applications. To cope with the increasing reliability concern for a broader range of microprocessors, a wide choice of dependability enhancement techniques should be developed.

1.2 Fault Tolerance Techniques

In order to achieve dependable computing, the first thing is to try to avoid fault occurrence. Careful design practices and new fabrication materials with better fault-immunity may lower the soft error rate. However, it is not sufficient to prevent the errors from occurring. No matter what efforts we make, faults can still occur. External disturbances cannot be completely blocked by enhanced shielding and packaging. Therefore, we need to design the processors to be tolerant of harmful impacts caused by soft errors. To achieve the desired level of dependability, *fault tolerance* approaches using redundancy must be employed.

The most common fault tolerance techniques (FTTs) are error-checking codes (ECCs) and hardware/time replication. For processor chip protection against soft errors, information in storage units such as caches is usually covered by ECCs [3], [8]. The protection codes are easily combined with regular structured memory arrays and check data bit errors requiring a small redundancy addition. The on-chip caches of many general-purpose processors are currently equipped with parity and/or SEC/DED codes [9], [10], [82], [83], [84], [11], [12]. The ECCs can also cover buses. ALUs may be protected by residue codes and illegal condition checks [13]. Other types of ECCs include m/n code, checksum, arithmetic code, cyclic code, etc. Unlike memory protection, integrity checking of random logic is very difficult due to irregularity. Traditional coding-based techniques may also be used for random logic at the circuit-level, but their applications and coverage are limited. Moreover, they may cause design delay and be prohibited due to performance penalties. At the system level, partial or entire computation can be repeated through the same or replicated processor modules for comparison [103], [104]. This approach can provide 100% error detection without introducing much design complexity. The major drawback of the system-level replication is high additional cost, and this usually restricts its use to general applications. Even if a sufficient die area can be budgeted for multiple processor modules, power dissipation issue can be an obstacle in many applications.

Signature monitoring is a technique to check the integrity of program execution and the flow of control by comparing reference and run-time information [14], [15], [16]. Traditional

signature monitoring schemes were developed at the system-level. A watchdog processor concurrently monitors the processor's behavior using signals on external address and data buses [17], [18]. Because of the scheme's inherent limitation in observability, the external signature monitoring for complex processors with built-in caches is not effective. The signature monitoring can be implemented fully in software [19], [20]. Instead of using a specialized hardware monitor, signatures are embedded in the program code and the control flow error is internally checked at every assertion point that is assigned by compiler's preprocessing. Since the monitoring consumes the processor's computing power, this approach degrades more than 30% of performance. In order to reduce performance impact, the ARC [21] technique exploits the processor's unutilized resources for error checking. It achieves high coverage in detecting the errors of some defined types, but the original resource usage of the base processor modeled is too low (on average 36% for integer unit and 17% for floating-point unit).

Idle resources can be also used for redundant execution of instructions. In the REdundant computation with Shifted Operands (RESO) technique [22], [23], [24], a function is recomputed with shifted operands for the verification using idle pipeline slots. Operand shifting makes it possible to detect permanent faults as well as transient faults in the computational functional unit. However, only computation instructions can be covered with this approach. In [25], [26], [27], re-computation approach is applied to superscalar processor models. The technique guarantees that every instruction in actual program flow is issued and computed twice by retaining decoded instruction in an internal buffer. Like the RESO, its coverage is still limited to execution phase of the pipeline.

Recently, thread-level fault detection schemes have been proposed for the simultaneous multithreading (SMT) architecture [31]. AR-SMT [32] and SRT [33] processors run two copies of the same program in separate thread contexts by dynamically partitioning resources, and then compare the outputs of each redundant instruction pair. While this approach checks the integrity of every instruction execution on the SMT processor, the performance penalty may be high unless enough idle resources are available. Executing each instruction twice results in 10$\sim$30% decrease in performance. The re-execution approach was also implemented in multiscalar architecture [34]. Despite a lot of processing resources, its performance degrades by 5$\sim$15%. In REESE approach on a superscalar processor model [35], spare hardware is strategically added to eliminate performance impact of re-execution. The study indicates that idle resources are often limitedly utilized for redundant execution, thus requiring a large addition of spare hardware.

Augmenting the commit phase of the processor pipeline with a checker was proposed for functional verification in microprocessor designs in [28], [29], [30], where a separately designed checker continuously monitors the processor's operations. Commit-time checking and logic optimization can provide area reduction for the checker processor. This approach however uses a heterogeneous checker that may require large design and verification overhead. Besides, the checker is unrealistically assumed to be fault free. Synchronization and implementation feasibility issues need more investigation in these proposals. In case of design errors, one can expect performance that of checker processor only.

For integration into modern microprocessors, FTTs need to achieve the goals of negligible performance impact, low cost, and high coverage. Although several processor integrity checking techniques have been proposed, achieving these goals at the same time is still an open question.

1.3 Our Research Goals

We have reviewed common techniques and identified issues and problems with them. Each FTT, in particular processor integrity monitoring (PIM) scheme, differs in the level and format of redundancy employed. The redundancy *level* represents the amount of extra operation time, hardware, and software devoted to integrity checking. The redundancy *format* indicates the degree of difference from original operation. The redundancy domain formed by these two factors. The coverage of a PIM usually improves as more redundancy is used. Each new redundancy format incurs complexity and increases time to design, implement, and verify. The general objective of this dissertation is to find proper design point in redundancy domain that achieves optimal coverage and complexity characteristics, while keeping the redundancy to a minimal level. Two key issues are 1) how to minimize various overheads of the redundant operation, and 2) how to maximize the protection coverage.

In order to realize our research goals, we have conducted an extensive study on the microprocessor dependability enhancement. The rest of the dissertation analyzes problems and presents our solutions as follows. Chapter 2 reports our fault injection experiments on the picoJava-II^{TM} processor, a commercial microprocessor from Sun Microsystems. Prior to any development of the FTT, it is fundamental to understand how much the processor is sensitive to transient faults. This provides us a guideline of strategic use of redundancy for enhancing dependability. Chapter 3 presents area efficient techniques for memory array protection. We also demonstrate our flexible design choices. Considering the fact that more than 50% of

modern processor chip area is devoted to on-chip cache memories, our techniques can make a significant contribution. Chapter 4 addresses the importance of initial data transfer from a lower memory to primary caches in multi-level caching system. Our technique assures correct data transfer utilizing idle bus cycles and existing information redundancy. Chapter 5 discusses challenges in protecting random logic. On-line monitoring technique using a signature cache is examined. Chapter 6 presents a system-level approach for superscalar processors. We prove that a duplex system can be realized with much lower overheads than the conventional duplex system. In Chapter 8 we makes some concluding remarks.

CHAPTER 2 Soft Error Susceptibility of the picoJava-II processor

Dependability is becoming an increasingly important quality measure of microprocessors in a wide range of applications. This chapter investigates the soft error sensitivity (SES) of the picoJava-II processor through software simulated fault injections in its RTL model. Soft errors are generated under a realistic fault model during program run-time. The SES of a processor logic block is defined as the probability that a soft error in the block causes the processor to behave erroneously or have an incorrect architectural state. The SES is measured at the functional block level. We have found that highly error-sensitive blocks are common for various workloads, while soft errors in many logic blocks rarely affect the computation integrity. Our results show that a reasonable prediction of the SES is possible by deduction from the processor's microarchitecture. We also demonstrate that the sensitivity-based integrity checking strategy can be an efficient way to improve fault coverage per unit redundancy.

2.1 Processor Fault Behavior

Understanding the processor's behavior in the presence of soft errors has a fundamental value in devising fault tolerant techniques, and fault injection methodologies can be used for that purpose [36], [37]. Faults are intentionally created in the processor with special software and/or hardware tools and the operations are monitored for erroneous effects. Ideally, the criteria of determining the protection requirement for a processor should include soft error rate and actual failure observation analysis, but mostly they are approximated. Even though reasonably accurate information is obtained by processor manufacturers, it is generally not revealed for long time. Soft error studies on commercial products have been rare in academia because of limited access to detailed processor models and/or experiment equipment. Therefore, it is not easy to have comprehensive knowledge of soft error characteristics for various microprocessors.

The fault injection is more often used to evaluate the effectiveness of the fault tolerance

mechanisms. A properly designed mechanism is expected to detect injected errors quickly and bring the processor state back to normal. While different options for the mechanism are tested to measure fault coverage, corresponding overheads also need to be examined. It is important that the error checking and recovery process does not degrade the processor's performance. The fault injection experiments facilitate all of these investigations.

This chapter presents a case study of soft error characterization using picoJava^{TM}-II, which is a microprocessor core developed by Sun Microsystems. The register transfer level (RTL) model of the picoJava-II became publicly available in 1999 to enable different groups of researchers to study, extend, and improve this commercial product. Our major effort here is to gain a good insight into the picoJava-II core's behavior under faulty environment. We employ a software simulated fault injection method and observe how much the core is susceptible to transient faults while processing programs. We also identify and characterize dominant factors that affect the processor's sensitivity to the faults. If this kind of investigation is combined with sufficient understanding of the processor architecture, the designers can significantly reduce overhead for integrity checking and maximize protection capability. This chapter substantiates such a synergetic case and provides a guideline of low-cost dependability enhancement.

The remainder of the chapter is organized as follows. The next section summarizes our literature survey on fault injection studies with emphasis on microprocessor behavior and identifies the position of our work. Section 2.3 provides a short introduction to the picoJava-II architecture. Experimental setups and challenges of the simulations are discussed in Section 2.4. The results are examined and important findings are addressed in Section 2.5. Section 2.6 provides a summary.

2.2 Soft Error Characterization through Fault Injection Methods

Transient faults can be injected into a microprocessor in many ways, and each method has different controllability of fault time and location, level of perturbation to the processor, and simulation time and cost requirement. Commonly used hardware methods are pin-level injection [38], [39], heavy-ion radiation [1], [40], and electro-magnetic disturbances [41], [42]. Recently, non-destructive laser has been also introduced [43], [44]. All of these methods closely imitate real fault situations, but they are usually expensive and applicable only after the physical chip is available. On the other hand, software fault injection is low-cost and can

be applied to programs and operating systems as well [46], [47]. This software method is classified into two kinds. The first class is *software-implemented* method, where the processor state or programs are modified during compile or run time and the injection takes place on real hardware. The other class is *simulation-based* method, where the processor, workload, and fault injections are all modeled in software simulation. In general, the latter is more flexible than the former as it provides better controllability of fault injection and observability of system behavior. In this chapter, we use a simulation-based approach.

Several tools have been developed to automate fault injection experiments and some tools are even able to analyze the observations they make. Extensive discussion on those is given in [36] and [37]. GOOFI in [49] is an object-oriented injection tool that is designed to be portable to different platforms. The efficiency of diverse fault injection tools are compared in [48]. An advanced tool reduces simulation time by conducting more than one injection simultaneously, and also supports event handling mechanisms and multiple system/fault models. Since fault simulation space is so large, it is always very challenging to obtain accurate behavior analysis in an acceptable time frame. Thus, a proper fault injection tool and technique should be selected for each target processor after careful examination. In our study, we adapt existing techniques, but modify them to be suitable for the picoJava-II core and our experimental environment. This will be explained in detail in Section 2.4.

Emulated fault models affect the fault manifestation. The most popular way to model the transient fault is logic inversion, where each fault flips the logic values temporarily. A study in [50] indicates that error behavior modeling is dependent on workload and hence various workloads need to be considered. However, there may also be common characteristics over different workloads that we can take advantage of in fault tolerant design. For example, SimR2K, a 32-bit RISC tested in [51], exhibited a very similar sensitivity pattern when faults were injected for several workloads.

More importantly, the effects of transient faults strongly vary with processor architecture and possibly fault injection methodology. In [52], a jet engine controller called HS1602 was upset by current and voltage transients. The results show that faults in the arithmetic unit are most likely to propagate and result in logic failure. In another experiment, RTL model of the IBM RT PC was injected with single-cycle inverted transient faults in [53]. About 60~70% of injected faults were overwritten. The study also reports that the attributes of the workload such as instruction types and control flow structures are good indicators of error behavior. Nevertheless, this claim has not been fully verified, and hardware organization might

have more direct impacts. Another software modeled 32-bit RISC, called TRIP, was tested using VHDL in [54]. The fault injection was performed by toggling the value of randomly chosen internal state element bits. While 34% of faults were overwritten at run-time, only 23% of faults were *effective*, i.e., the faults resulted in processor failure. It should be noted that processors are capable of masking out some faults without any intended protection mechanism.

Other systems or processors investigated for error behavior include MC6809E with heavy-ion radiation and power supply disturbance [40], MC68000 with device-level simulation [55], SPARC1 system with physical injection [56], MC88100 with combination of software-implemented and simulation-based fault injection [57], MC68302 with VHDL simulation [58], and MARK2 with simulated stuck-at and open-line faults [59]. All these studies support the fact that each processor has a distinct level of sensitivity to soft errors, and therefore, a new design requires separate dependability evaluations and may be engineered for lower sensitivity.

Once soft errors occur in the logic blocks of a processor, their propagation nature is mainly defined by the architecture and workload of the processor. On the other hand, how often the soft error appears, *transient upset rate*, is affected by the fabrication process and circuit technology. More upsets mean higher probabilities of soft error occurrence. In [60], the same heavy ion was individually radiated into three units of a ERC32 processor, but upset rates were different because the units employed diverse circuit types. Errors were observed mostly in the register file and some in the combinational logic. Circuits of the integer unit were more susceptible to the ions than those of floating point and memory control units. Another radiation testing on 486DX4 microprocessors [61] shows that different implementations of a common processor architecture result in susceptibility variation. When six 486DX4 processors from AMD and Intel were bombarded by radiation beams, AMD's chips were more susceptible than Intel's. In the experiment, the feature size of AMD's was smaller than Intel's.

This dissertation focuses on what impacts soft errors make on the computation of the picoJava-II core rather than how often faults generate the soft errors in the processor. Thus, our study is independent of implementation and process technology. It can be a base for the development of architectural solutions.

2.3 PicoJava-II Core Architecture

The picoJava-II is a microprocessor core uniquely designed for directly executing Java bytecode instructions defined by the Java^{TM} Virtual Machine (JVM) [62] in hardware, and it is used

for cost-sensitive embedded applications [63]. This low-cost hardware implementation of JVM offers high performance without the need for just-in-time compilers. The core is equipped with a six-stage RISC pipeline and instruction folding capability (a process of loading and executing an instruction in a single cycle). With proper compilation, the core can also process programs written in legacy C/C++ language comparable in speed to other RISC processors. Chips based on the picoJava-II are well suited for a wide range of information appliances such as digital set-top boxes, Internet TVs, cellular phones, personal digital assistants, automotive communication devices, etc. As the use of such products increases, their malfunctions due to transient faults may cause serious loss of time, money, or even worse. Dependability will become a more important quality measure for the products. Therefore, sufficient estimation and enhancement of dependability are imperative.

The picoJava-II core we use in our study is a soft intellectual property (IP) as opposed to hard IP under the sun community source licensing program [64]. The soft IP described in Verilog, a popular hardware description language for logic design, specifies the detailed implementation of the picoJava-II core and completely simulates the real chip. To make the chip more suitable for a particular application, some features of the core can be reconfigured. However, we consider a core with the standard features in the original IP.

A full description of the picoJava-II core architecture is presented in [65]. Figure 2.1 illustrates our floor plan for an implementation of the picoJava-II. The area ratios between functional unit blocks (FUBs) are approximated by using picoJava-II synthesis results from [66]. When the area is measured in terms of 2-input NAND equivalent gates, more than 69% of the chip is devoted to cache memory arrays and control logic. To assist the core behavior analysis, we briefly discuss the functionalities of its major FUBs: (1) integer unit (IU); (2) instruction cache unit (ICU); (3) instruction cache tag (ITAG); (4) instruction cache RAM (ICRAM); (5) data cache unit (DCU); (6) data cache tag (DTAG); (7) data cache RAM (DCRAM); (8) floating point unit (FPU); (9) stack manager unit (SMU); (10) power-down, clock, and scan unit (PCSU); (11) bus interface unit (BIU); and external memory controller (MEMC).

The IU decodes and executes instructions from an instruction buffer (I-buffer). It forwards floating point instructions to the FPU and communicates with the DCU for data. The IU consists of a 32-bit ALU and shifter, a multiply/divide unit, a microcode ROM implementing multi-cycle instructions, registers, a 64-entry stack cache (SC), and trap generation/dependency checking/forwarding logic. The ICU controls a 16-KB direct-mapped instruction cache (I-

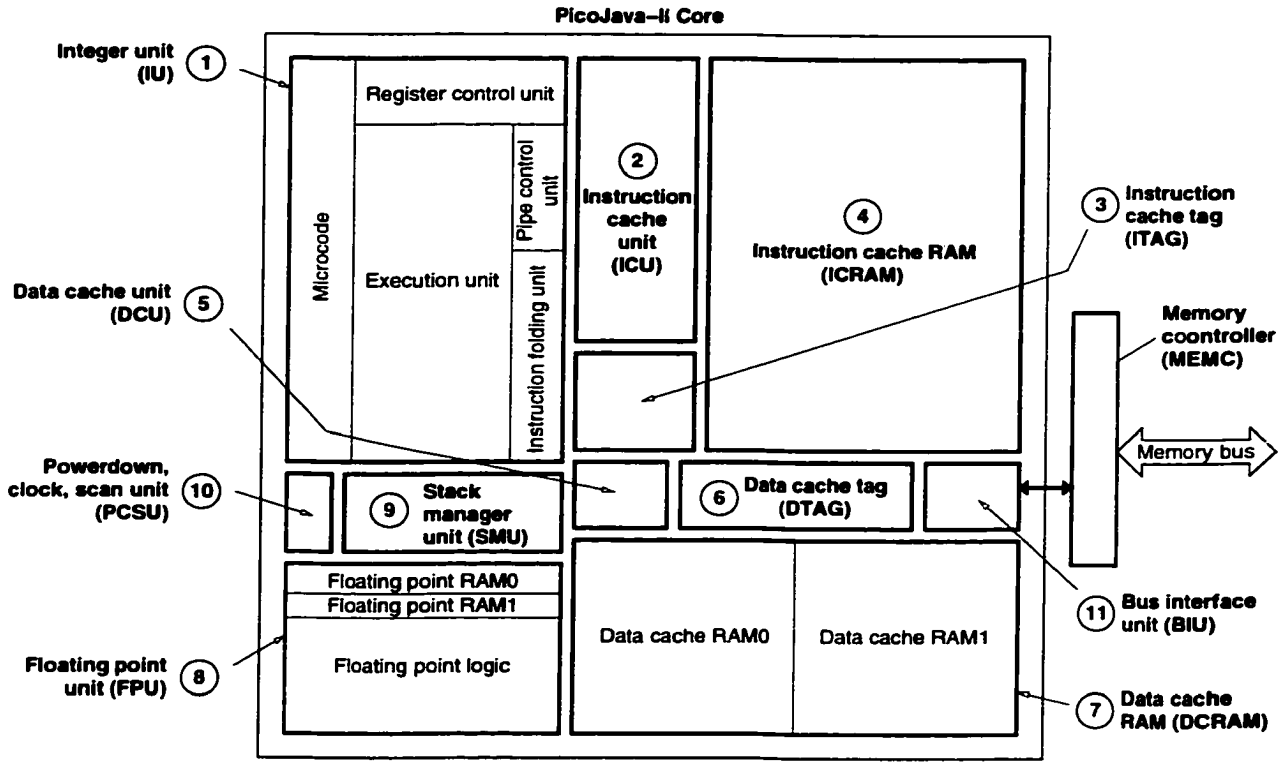


Figure 2.1 Block diagram of the picoJava-II.

cache) and the I-buffer. It fetches and dispatches instructions to the decode unit of the IU. The ITAG contains 1024 19-bit tags. Each tag corresponds to its associated line of 4 words in an ICRAM. The ICRAM holds instructions in a RAM of 2048 entries by 64 bits. The DCU handles souring and sinking of data from/to a data cache (D-cache) for load and store instructions. The D-cache consists of a 1024×24-bit DTAG and a 4096×32-bit DCRAM, configured for 16-KB with 16-byte line, 2-way set associative, write-back, and write-allocate. The FPU executes the floating point instructions. The SMU handles overflow and underflow conditions of the SC. The PCSU integrates power management, clock generation, system reset, scan, and test. The BIU is the interface between the core and external world via the MEMC. The MEMC is the interface between the BIU and external devices such as memory and I/O.

2.4 Experimental Setup

We conducted a set of transient fault injection simulations on the RTL model of the picoJava-II processor. This model has been synthesized, validated, and optimized by sev-

eral commercial vendors and universities. Based on a fault model we defined, the faults were simulated entirely in software while each application program was running. Our fault model basically follows the conventional signal inversion approach, but it is not restricted to single-bit failure. The probability of a fault occurrence is uniformly distributed over operation time and logic location. A fault in a logic block is manifested as a logic value toggle of its output signals, from 0 to 1 or from 1 to 0. Multiple signal bits can be corrupted by a single fault.

Fault injection locations (FILs) in the processor are determined on a minimal logic block basis. Each FUB shown in Figure 2.1 is divided into sub-blocks after examining error propagation paths with test generation rules. If logic gates in a FUB have common fault effects on an output signal of the FUB, i.e., fault equivalent gates, they are grouped together and treated as a single FIL. Consequently, a FIL is a sub-block responsible for producing an output signal of a FUB, and a fault in the FIL means a soft error in the corresponding output signal.

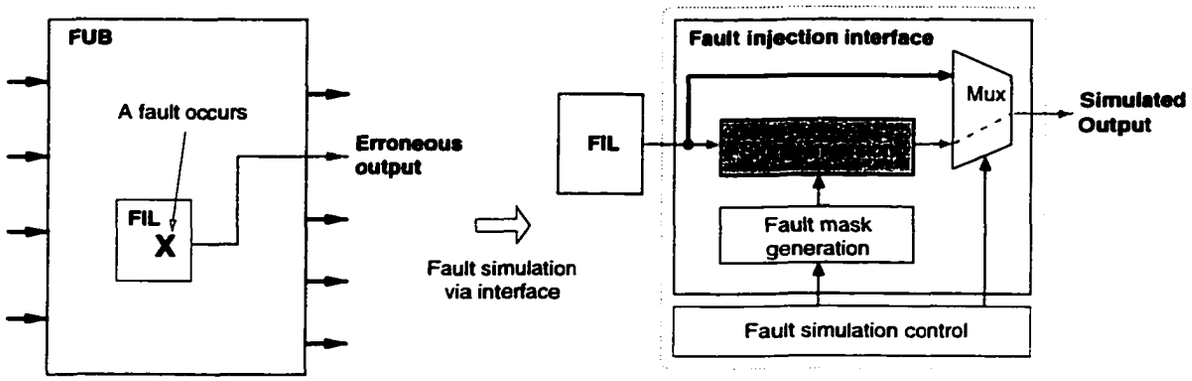


Figure 2.2 Transparent software interface for fault injection.

The fault injections are carried out through software interfaces appended to the processor. Figure 2.2 depicts the block diagram of our fault injection interface integrated with a FIL. An interface is placed at each output port of the FUB, and it takes the output signal and produces a simulated output signal under the fault simulation control. The simulation control is another software module that allows us to direct the fault type and injection timing and duration. The fault mask specifies bit positions for the signal inversion. The total number of error bits per injection is chosen randomly with a minimum value of 1. Although the injection interfaces are embedded in the processor simulator, they are completely transparent to any other component and effective only in the case of fault simulation.

Table 2.1 lists all FILs identified for the picoJava-II FUBs and describes their output

Table 2.1 (a) The FILs of the IU and I-cache.

FUB	FIL	Output signal Name / Width		Description	Destination FUB(s)
IU	1	iu_addr_e	32	Address to the I-cache & D-cache	ICU, DCU
	2	iu_br_pc	32	Branch or trap target PC	ICU
	3	iu_brtaken_e	1	Branch instruction is taken	ICU
	4	iu_icu_flush_e	1	Flush the I-cache	ICU
	5	iu_psr	32	Processor status register	ICU, DCU, SMU
	6	iu_shift_d	8	The number of data bytes shifted from the IFU	ICU
	7	iu_data_e	32	Bypassed IU data to the I-cache and D-cache	ICU, DCU
	8	iu_kill_dcu	1	Terminate outstanding DCU operations	DCU
	9	kill_inst_e	1	Kill DCU instruction	DCU
	10	iu_special_e	1	Special instruction	DCU
	11	iu_dcu_flush_e	3	Flush instructions	DCU
	12	iu_inst_e	8	Load and store instruction	DCU
	13	iu_zero_e	1	A zero line instruction	DCU
	14	iu_data_in	32	Stack base-address	SMU
	15	iu_sbase_we	1	Write enable for iu_data_in	SMU
	16	iu_optop_din	32	Value of top of stack pointer register (OPTOP)	SMU
	17	iu_optop_int_we	1	Write enable for iu_optop_din	SMU
	18	ret_optop_update	1	Return instruction updating OPTOP	SMU
	19	iu_smu_flush	1	Kill all loads and stores	SMU
	20	iu_rf_dout	32	Spill data from the IU	SMU
	21	iu_smiss	1	SC write miss	SMU
	22	iu_smiss_addr	32	The address for the SC write miss	SMU
	23	iu_smiss_data	32	The data for the SC write miss	SMU
	24	iu_powerdown_op_e	1	The IU executes powerdown instruction	PCSU
	25	iu_rs1_e	32	The first operand for FPU	FPU
	26	iu_rs2_e	32	The second operand for FPU	FPU
	27	fpop	8	Java floating point opcode	FPU
	28	fpop_valid	1	Valid fpop input	FPU
	29	iu_kill_fpu	1	Terminate the fpop	FPU
	30	hold_fpu	1	Freeze an FPU operation	FPU
ICU	31	icu_data	56	Top 7 bytes of the I-buffer	IU
	32	icu_drty	7	Dirty bits of the 7 entrees in the I-buffer	IU
	33	icu_vld_d	7	Valid bits of the 7 entrees in the I-buffer	IU
	34	icu_length_d	28	Lengths of instructions in the I-buffer	IU
	35	icu_pc_d	32	PC of the first instruction of the I-buffer	IU
	36	icu_hold	1	The ICU is unable to service request	IU
	37	icu_din	32	Data to be written to the ICRAM	ICRAM
	38	icu_ram_we	2	Write enable to ICRAM	ICRAM
	39	icram_powerdown	1	Power-down the ICRAM and ITAG	ICRAM, ITAG
	40	icu_addr	14	I-cache address used	ICRAM, ITAG
	41	icu_tag_in	18	Data to be written to the ITAG	ITAG
	42	icu_tag_vld	1	Valid bit value for the ITAG	ITAG
	43	icu_tag_we	1	Write enable to the ITAG	ITAG
	44	icu_in_powerdown	1	The ICU is ready for standby mode	PCSU
	45	pj_icureq	1	Request instructions from the BIU	BIU
	46	pj_icusize	2	The size of the I-cache transaction	BIU
	47	pj_icutype	1	The type of the I-cache transaction	BIU
	48	pj_icuaddr	32	Physical address for the instruction request	BIU
ITAG	49	itag_dout	18	Tag data from the ITAG	ICU
	50	itag_vld	1	Valid bit from the ITAG	ICU
	51	ic_hit	1	I-cache hit	ICU
ICRAM	52	icram_dout	64	Data from ICRAM	ICU

Table 2.1 (b) The FILs of the D-cache and the remaining units.

FUB	FIL	Output signal Name / Width		Description	Destination FUB(s)
DCU	53	iu_data_vld	1	Data on dcu_data is valid IU data	IU
	54	dcu_data	32	Data from the D-Cache	IU, SMU
	55	iu_stall	1	Stall the IU pipeline	IU
	56	dcu_err_ack	3	Error acks due to data access exception	IU
	57	dcu_in_powerdown	2	The DCU is ready for standby mode	PCSU
	58	dcu_din_e	32	Data to be written to the DCRAM	DCRAM
	59	dcu_ram_we	4	Write enable to the D-cache	DCARM
	60	dcu_pwrdown	1	Power-down the DCRAM and DTAG	DCRAM, DTAG
	61	dcu_stat_addr	13	Address for writes to status register and DCRAM	DCRAM, DTAG
	62	dcu_bank_sel	2	Select bank of the DCRAM	DCRAM
	63	dcu_bypass	1	Bypass the DCRAM	DCRAM
	64	dcu_tag_in	19	Data to be written to the DTAG	DTAG
	65	dcu_stat_out	5	Data to be written to status reg. of the DTAG	DTAG
	66	dcu_set_sel	1	Select set of the DTAG	DTAG
	67	wb_set_sel	1	Write buffer set select	DTAG
	68	dcu_tag_we	1	Write enable for the DTAG	DTAG
	69	dcu_stat_we	5	Write enable for status reg. of the DTAG	DTAG
	70	dcu_addr_out	32	Address of data to be accessed in the D-cache	DTAG
	71	smu_stall	1	Stall the SMU pipe due to a cache miss	SMU
	72	smu_data_vld	1	Data on dcu_data is valid SMU data	SMU
	73	dcu_smu_st	1	The SMU stores in the cache-stage	SMU
	74	pj_dcureq	1	Request data from the BIU	BIU
	75	pj_dcusize	2	The size of the D-cache transaction	BIU
	76	pj_dcutype	3	The type of the D-cache transaction	BIU
	77	pj_dcuaddr	32	Physical address for the data request	BIU
	78	pj_dataout	32	Data to be written to the external device	BIU
DTAG	79	tag_dout	19	Tag data of the selected set	DCU
	80	dtg_stat_out	5	Status bits of DTAG	DCU
	81	hit0	1	D-cache hit in set 0	DCU
	82	hit1	1	D-cache hit in set 1	DCU
DCRAM	83	dcram_dout	64	Data from the DCRAM	DCU
FPU	84	fp_rdy_e	1	The FPU is ready to accept new request	IU
	85	fpu_data_e	32	The result of the FPU operation	IU
SMU	86	smu_rf_din	32	Fill data to the SC	IU
	87	smu_rf_addr	6	The SC for reads and writes	IU
	88	smu_we	1	Write enable to the SC	IU
	89	smu_sbase	30	Updated base stack-address	IU
	90	smu_sbase_we	1	Write enable for smu_sbase	IU
	91	smu_hold	1	Hold the IU pipeline	IU,DCU
	92	smu_addr	32	Address for spills/fills & memory protection check	IU,DCU
	93	smu_st	1	A D-cache store request during spills	IU,DCU
	94	smu_ld	1	A D-cache load request during fills	IU,DCU
	95	smu_data	32	Data for the D-cache writes	DCU
	96	smu_na_st	1	A nonallocate store	DCU
PCSU	97	pj_nmi_sync	1	Synchronized nonmaskable interrupt	IU
	98	pj_irl_sync	4	Synchronized interrupt request line	IU
	99	pcsu_powerdown	1	Power-down (standby mode) request	ICU, DCU, FPU
BIU	100	pj_dcuack	2	Bus acknowledgment for an D-cache transfer	DCU
	101	pj_icuack	2	Bus acknowledgment for an I-cache transfer	ICU
	102	pj_datain	32	Data from an external device	ICU, DCU
	103	pj_data_out	32	Data written to an external device	MEMC
	104	pj_address	30	Address of a data access to an external device	MEMC
	105	pj_size	2	The size of a data transfer	MEMC
	106	pj_type	4	The type of a data transfer	MEMC
	107	pj_tv	1	The indication of a valid cycle on the bus	MEMC
MEMC	108	pj_data_in	32	Data read from an external device	BIU
	109	pj_ack	2	The bus acknowledgment for a transaction	BIU

signals with destined FUBs. One can further break down each FIL into smaller sub-FILs at a lower level of the RTL model hierarchy, but it does not provide significant advantages in understanding fault behavior despite the need of a large increase in evaluation time. For example, 32 bit slices for an address adder in FIL 1 can be separately examined, but additional knowledge we expect to gain from that is small. Each FIL can include both random logic and memory arrays, or only one of them. When the fault is injected into the FIL, its function determines the type of soft error being generated such as a control error, data error, etc.

Four application programs implementing different algorithms were used for benchmarking workloads as follows. *Bubble* implements an elementary bubble sort method for sorting 64 integer numbers. The algorithm involves comparing and exchanging elements to properly position in a data structure. *Crypt* performs encryption and decryption using IDEA (International Data Encryption Algorithm) on an array of N bytes. This Java kernel program is a part of the Java Grande Forum Benchmark Suite [67]. N was 800 in our study. We also used an optimized Java version of the well known linpack benchmark. *Linpack* solves an $N \times N$ linear system using LU factorization followed by a triangular solve. It measures floating point performance with a numerically intensive test. Our problem size was 25×25. *Queens* finds all the solutions of the Eight Queens problem on an 8×8 board using a recursive algorithm. We believe these programs carry out frequently used operations in the applications of the picoJava-II. All benchmarks were tested on both the picoJava-II processor and other JVM platforms for a cross-check.

If a fault is injected at a certain time of program execution, its effects may appear immediately or many cycles later. Thus, observation through the end of execution may be necessary in order to check all possible scenarios of that particular fault. This approach however is not suitable for testing many faults because each fault injection takes a full program execution time, which can be a few days in the RTL simulation. To obtain statistically significant results, a large number of fault cases should be considered. Thus, minimizing simulation time per injection is necessary.

With more simulation control and a slightly lowered level of observation detail, several faults can be examined during a single execution. Figure 2.3 illustrates how we performed the fault injection and made observation in a targeted FIL on the picoJava-II. At the beginning of each simulation run, the RTL simulator invokes another processor simulator, called the Instruction Accurate Simulator (IAS), which models all functionality of the picoJava-II by instruction boundaries. The architectural states of both simulators are identical after the commitment of

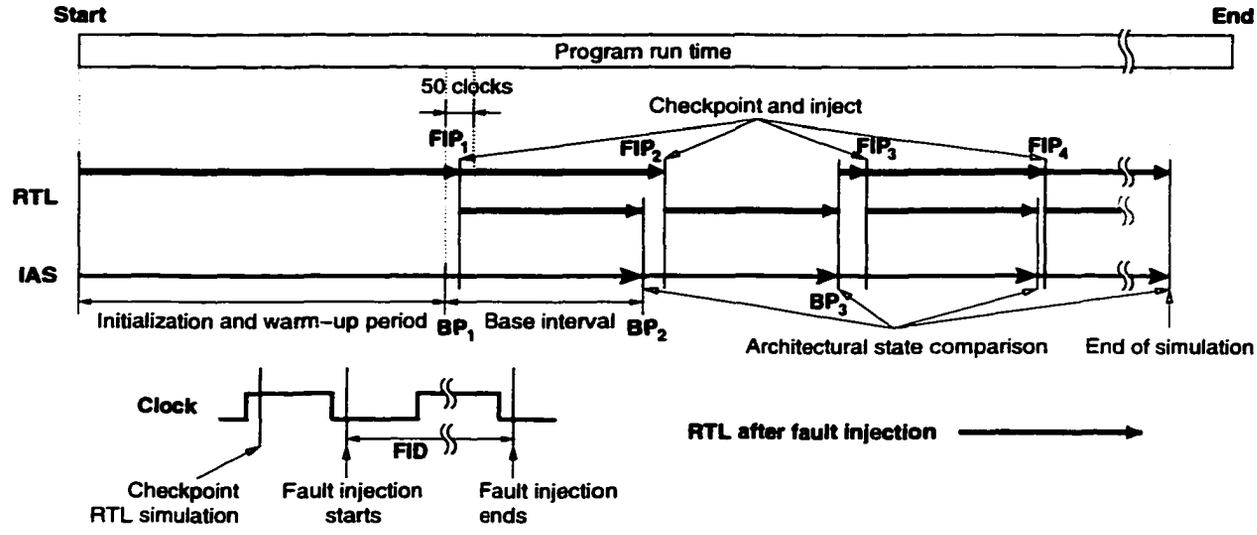


Figure 2.3 Fault injection and observation timing diagram.

every instruction under the fault free condition. The primary purpose of the IAS is to serve as a golden run for monitoring the RTL simulation when the fault is active. Fault injection points (FIPs) at which faults are injected into the processor are considered after the initialization and warm-up period. The FIP_1 is randomly picked from a window of 50 clock cycles starting from the first basepoint (BP_1). At the FIP_1, the RTL simulation is checkpointed, i.e., saving the current state of the simulated processor, during the first half of the clock period. When the clock goes to low, the fault injection begins and lasts for a predefined interval, denoted by fault injection duration (FID). A single clock cycle may be enough for the injected fault to be latched as a soft error. The longer the fault is resident in the FIL, the more likely it is to affect the processor. Given this behavior we chose 1 and 10 clock cycles for short and long FIDs, respectively.

After the fault injection at FIP_1, the simulation is monitored up to the next base point (BP_2), and then the architectural state of the RTL simulator is compared with the state of the IAS simulator. If the fault has been effective, the states mismatch. In this case, the RTL simulator rolls back to FIP_1 using the checkpointed state and the IAS is paused until the RTL reaches BP_2 with no fault injection. Otherwise both simulators proceed to FIP_2. The same procedure is repeated for subsequent fault injections. The more FIPs tested for a FIL, the higher accuracy in estimation. We chose 100 FIPs during a benchmark run for a single FIL.

In our simulation, every single fault injection created a soft error in the FIL. The effects

of different occurrence timings of soft errors were examined. We define *soft error sensitivity* (SES) of a FIL as the probability that a soft error in the FIL causes the processor to behave erroneously or have an incorrect architectural state. The outcome of each fault injection at a FIP is classified into four cases: 1) no harmful impacts are made to the processor; 2) the program running on the processor hangs; 3) the system crashes before the simulation reaches the next base point for the state comparison; or 4) the simulation continues to the base point, but the architectural state is corrupted. Thus, the SES of a FIL is computed by $\frac{\sum_{i=2}^{4} Case\ i}{number\ of\ faults}$. We measured the sensitivities of all 109 FILs in the picoJava-II.

It is important to use a sufficiently large base interval so that the soft error has enough time to manifest as a noticeable failure. We set the base interval to 600 clock cycles. It should be noted that the soft error may not generate any failure or may be overwritten. Other metrics such as error latency and propagation pattern are helpful in fault characteristics analysis. However, we focused only on the SES collection and analysis since, given our limited resources, it still enables us to obtain a reasonable level of understanding of the picoJava-II.

2.5 Soft Error Sensitivity of the PicoJava-II

In this section, we present and analyze the SES results for four workloads. The results illustrated together are based on the common simulation parameters unless otherwise specified. The error of our SES estimate is less than 0.098 for a 95% confidence level. The architectural significance of FILs are reviewed and their relations with the SES are considered. We also point out what we can benefit from the SES information in devising an integrity checking scheme for the picoJava-II, and evaluate how well the techniques used in current microprocessors reflect the soft error behavior.

2.5.1 Integer unit

Figure 2.4 depicts the distribution of effective faults in FILs of the IU. The FID is 1 clock cycle. The most apparent characteristic in this figure is that only a few FILs are highly sensitive to the faults. There are many FILs whose corrupted output signals do not stimulate any erroneous behavior of the processor. We can also note that sensitive FILs are mostly common for different programs and the types of the impacts are similar. Clearly, a faulty signal is effective only if it plays an active role in the processor's operation. If the signal is not stored, the window when the processor is vulnerable to the fault is only 1 clock period. Even

though the signal can propagate to other components, there is still a chance that the error is overwritten before it creates any problem. The results indicate that many sub-blocks in the IU infrequently become a critical part of the operation.

Less than 10 sub-blocks overall are considerably susceptible. FIL 1 produces the address of the I-cache and D-cache. An incorrect address here means a memory access to a wrong location. On a write, the memory state may become faulty. On a read, unexpected data or instructions may be delivered to the processor. Consequently, a soft error in this sub-block is critical. However, it is only occasionally true because the cache memories are not always accessed. The erroneous write may occur at an invalid location and never get used. Although it is not very likely, data items or even instructions from multiple locations may be the same. Thus, reading from any of these locations is identical. In short, there are many conditions that can stop or trigger the malicious actions of the soft error in FIL 1. We have observed that the cache address error causes system crashes when it is effective, and the SES of the corresponding sub-block is about 0.27, 0.08, 0.06, and 0.12 for *Bubble*, *Crypt*, *Linpack*, and *Queens* respectively. The error margin for 95% confidence level is ± 0.09, ± 0.05, ± 0.05, and ± 0.06, respectively.

Memory address checking is performed to some extent in conventional processors. The operating system can check the address range of each memory access, yet this cannot detect an error within an accessible region. The address generator, i.e., an adder, can be protected with parity prediction, residue code, or similar kind. Duplication may be too costly for low-end systems. The frequency of the cache access depends on the program, and it may increase in superscalar processors. In any case, relative importance of this sub-block is high, and thus, it is worth covering if possible.

FIL 3 is a part of branch logic informing the ICU and the pipeline that a branch should occur after the instruction in the execution stage. When it is faulty, the control flow of program may change. It can affect timing of the pipeline as well. As soon as the execution deviates from the correct flow, the processor state becomes erroneous, which can lead the processor to crash within the base interval (600 clock cycles). As a result, the SES goes up to 0.86. This sub-block is a critical component of the IU. FIL 6 is another critical part as it can affect the program counter (PC) and instruction shifting to the I-buffer. Since this sub-block is also concerned with the control flow, its erroneous behavior is similar to that of FIL 3.

Check code-based protection cannot be easily applied to protect random logic in FUBs such as FIL 3 and FIL 6. Unlike memory arrays, the relation between input and output

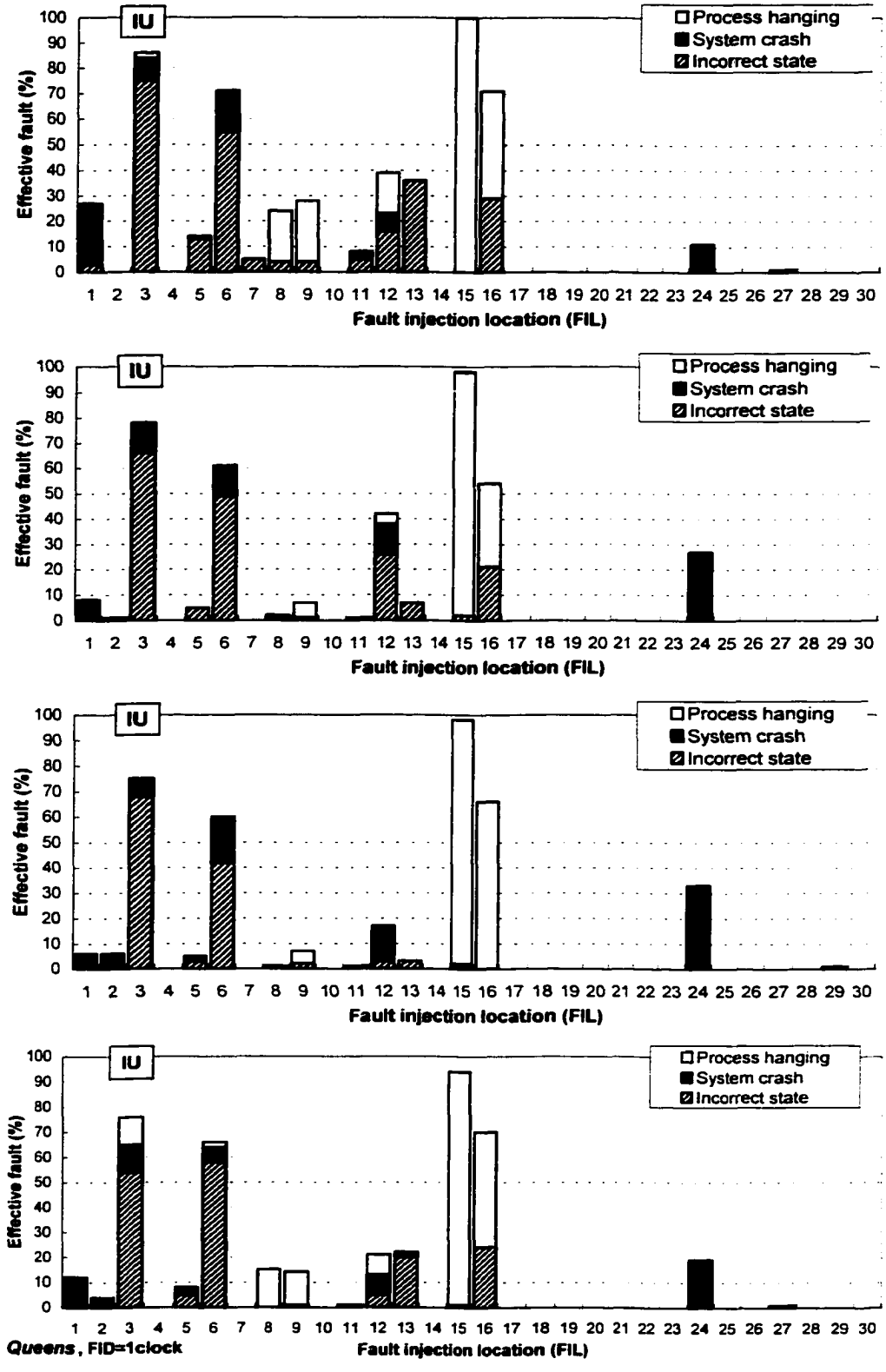
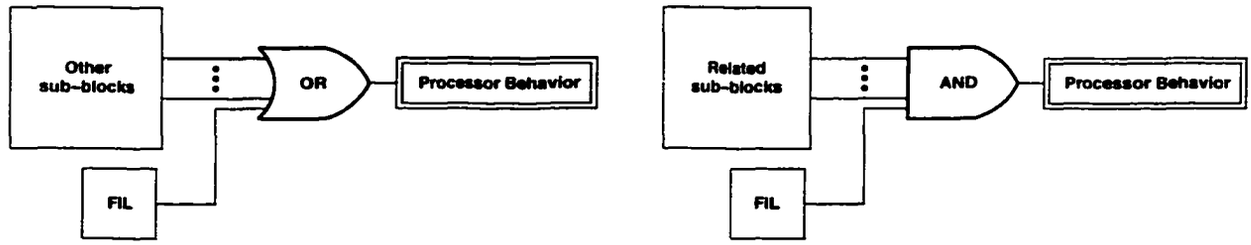


Figure 2.4 The SES of the integer unit (IU) when FID = 1.

signals dynamically vary. A simple approach is to duplicate the logic. The replication can be considered at any level of logic block hierarchy. FUB-level duplication is the simplest, but area overhead is large. Critical sub-blocks can be selectively duplicated to lower the overhead.

An error in FIL 8 or FIL 9 can result in an unexpected termination or continuation of the D-cache access instruction and its operation. FIL 12 controls the size, type, cacheability, and endianness of the D-cache access. FIL 13 signals the D-cache to fill a particular cache line with 0's. All these sub-blocks are critical only when the D-cache is in action. Among the IU sub-blocks, FIL 15 is the most susceptible to the error. This is because the earliest entry of the SC is always corrupted if its output signal is toggled. If this happens, the process is very likely to be suspended. It should be noted that the SES of FIL 15 for *bubble* is 1. This is a *first-protect* portion of the processor. FIL 16 has similar impacts as it is responsible for the top of the stack pointer. The picoJava-II supports an additional *standby* mode for low power management. The last critical sub-block FIL 24 may assert a false signal to the PCSU, which eventually puts the I-cache and D-cache in standby mode. In this case, the system crashes immediately. Converting standby mode to active mode does not degrade computation integrity.



(a) (b)

Figure 2.5 Possible logical positions of a FIL and their impacts on the sensitivity.

So far we have inspected relatively critical sub-blocks in the IU. Their functions are closely connected with essential steps of instruction executions. They are frequently used or placed in a logical position such that their malfunctions can often be realized as a failure. Figure 2.5 illustrates two logical positions that a FIL can be situated with other sub-blocks in the processor. The OR and AND are not binary logic gates, but rather conceptual blocks that may consist of several processor components. Figure 2.5a is the case where the processor behavior is directly governed by the FIL independently of other sub-blocks. It has its own impact on the system behavior. On the other hand, Figure 2.5b represents the case where the FIL is effective

only with the assistance of other sub-blocks. Unless all the related sub-blocks are active, the faulty FIL has no meaning. FIL 15 is an example of the first case. If a FIL falls in the second case, its SES depends on the active period of the other related sub-blocks.

In our experiment, there are many IU FILs that seemed unaffected by the fault injection for both FIDs tested (1 and 10). This can be explained by the functionalities of the sub-blocks as described in Table 2.1. Non-critical sub-blocks are generally involved in infrequently occurring operations and their logical positions are mostly the case of Figure 2.5b. For example, erroneous FIL 14 appears as a failure only if FIL 15 asserts a write enable signal, which is a rare event. Although some FILs can be the case of Figure 2.5a, their errors may not be serious. An example is that faulty FIL 4 flushes or invalidates an I-cache line, but this only produces an additional memory access. However, not flushing an I-cache line related with self-modifying code support or flushing a dirty line in the D-cache (FIL 11) can alter the processor state. FIL 30 can stall the FPU for extra cycles, but the processor maintains the computation integrity. Similar reasonings can be made for other sub-blocks.

Although many FILs never exhibit any effective fault in our results, it does not mean that they are unnecessary components. It simply shows that randomly selected erroneous periods of the FILs did not overlap with their active cycles or their faults have nothing to do with the correctness of the computation. When the FID was increased to 10 clocks, more effective faults were observed. Figure 2.6 presents the SES results in that case. A FID of 10 is an extreme example modeling a strong noise hit that prolongs the circuit's recovery time. FILs 4, 10 and 28 start to show their susceptibility. Interestingly in FIL 15 for *Linpack*, the SES is slightly lower than the case with the FID of 1. A possible explanation of this is that a faulty signal at an earlier cycle corrupted the first entry of the SC, and then a faulty signal at a subsequent cycle unintentionally corrected it back to normal. Different FIDs change the type of failure in a small degree. In order to check if a further increase in the FID makes a difference in non-critical sub-blocks, we injected them with 500-clock long faults for *Crypt* and monitored using a 800-clock base interval. Figure 2.7 plots the sensitivities of non-critical sub-blocks in all FUBs. Many sub-blocks become somewhat sensitive.

When considering a protection plan for the IU, our SES results can serve as a standard for ranking the sub-blocks in order of importance. We have found that sub-blocks with two opposite characteristics, ***highly*** susceptible and ***seldom*** susceptible to the soft error, do not vary with workloads. Moreover, this situation remains for faults with a longer FID. From our data, we believe that checking the top few critical sub-blocks alone enhances the integrity of

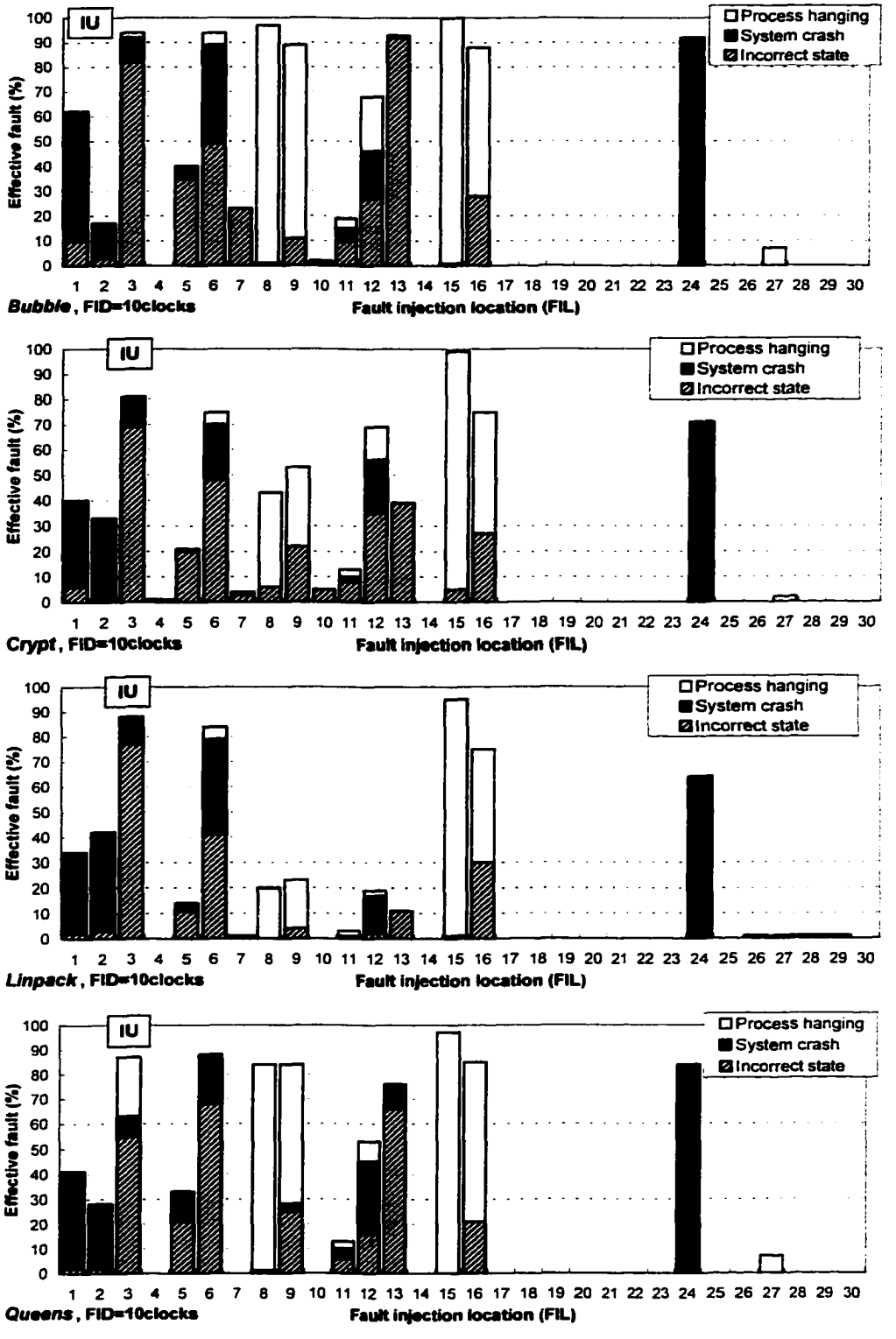


Figure 2.6 The SES of the integer unit when FID = 10.

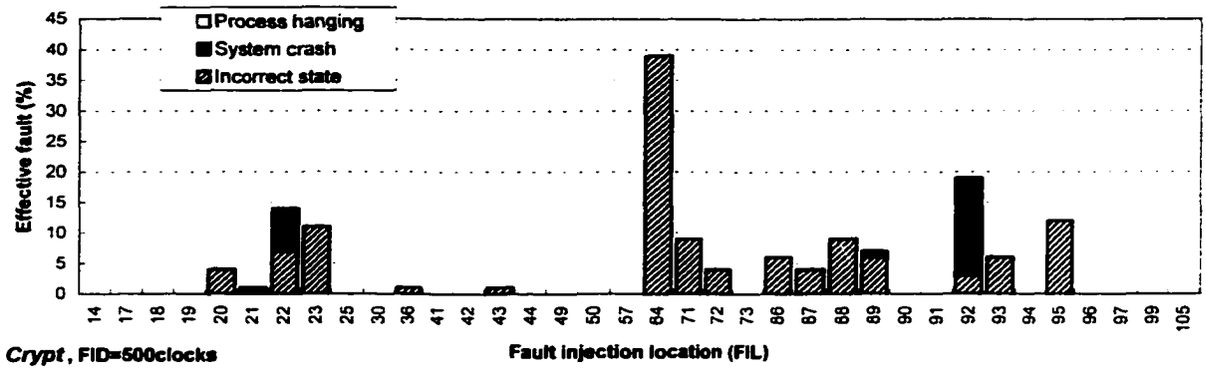


Figure 2.7 The SES check for non-critical sub-blocks by extending the FID to 500 clocks.

the IU significantly. This is a similar concept exploited in cache memory, which contains the most frequently used data and covers most of the memory access requests. Depending on the logic type of the sub-block, diverse fault handling techniques can be employed. If we take into account design and verification complexity, duplication is a fair choice for random logic. We claim that the protection plot based on the SES is a very efficient method to maximize the fault coverage with limited resources.

2.5.2 Instruction cache

Figure 2.8 shows the SES data measured for the I-cache. For each FIL, the two columns represent the SES when FID is 1 and 10 clocks, respectively. Critical sub-blocks are fewer than the IU. FIL 31 represents instructions stored at the top 7 bytes of the I-buffer. Its error results in the execution of incorrect instructions, but the instructions are effective only when their valid bits are set. In addition, they are removed on a branch or trap, rendering them inactive. Even though this sub-block is a memory array, we first injected the faults for a single clock to model transient faults in its random logic portion. If the faults occur in the memory cells, the data error stays until overwritten. This situation was moderately tested with a FID of 10. As expected, the SES of FIL 31 greatly increases with a longer fault duration. Any consuming instructions by the IU are supplied by the I-buffer. Errors in FIL 32 cause instruction state transitions. FIL 38 is critical because it allows instructions in the ICRAM to be modified. FIL 52 is active when its data bytes are transferred to the I-buffer. Any corruption in the I-buffer is nullified by a flush.

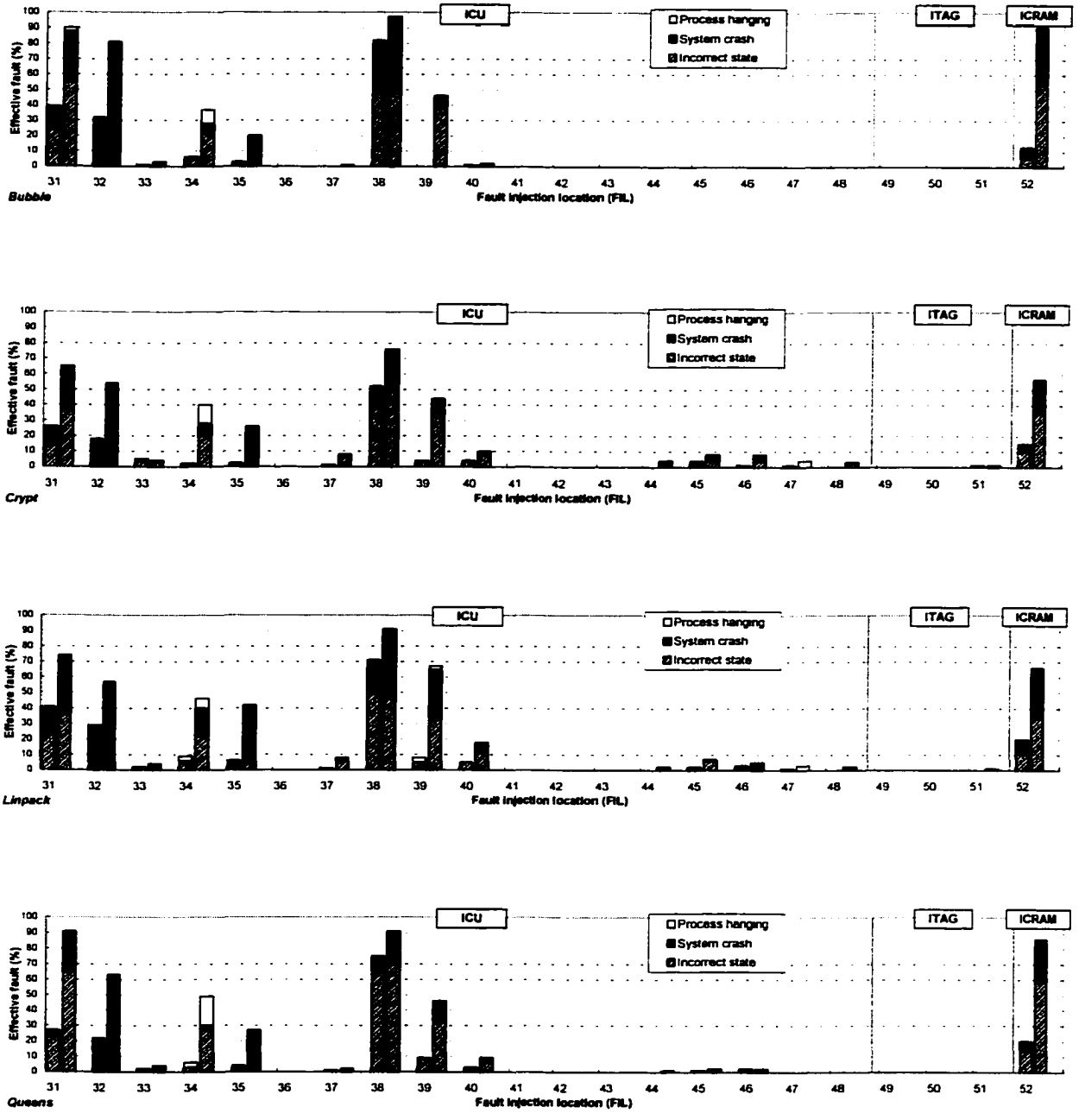


Figure 2.8 The SES of the I-cache.

Combining Figures 2.7 and 2.8, distinctively non-critical sub-blocks of the I-cache are FILs 41, 42, 44, 49, and 50. FILs 41 and 42 are effective only when FIL 43 is active. This models the case in Figure 2.5b. If their errors propagate to the ITAG, they are more likely to appear as a false I-cache miss, in which the processor fetches instructions from the memory instead. The situation is similar for FIL 49 and FIL 50, and therefore, they become hardly sensitive to the faults. The PCSU shuts off the clock for standby mode only if both the ICU (FIL 44) and DCU (FIL 57) signal that they are ready. A false assertion by FIL 44 alone is automatically masked.

For the I-cache protection, instruction memory arrays such as the I-buffer and ICRAM should be considered first. Also, control logic for them needs to be covered before the ITAG and its control logic. The tag portion of the I-cache is much less error-sensitive than the data portion. In conventional processors, memory arrays for both tag and data are often protected with ECCs, but surrounding control logic is left unprotected. Control logic errors have a short lifetime unless they are propagated to a storage component, whereas data errors in memory arrays remain active for a longer period. In this context, it is logical to protect memory arrays. However, highly susceptible control logic is as critical as instruction memories.

2.5.3 Data cache

The D-cache handles the processor's data write requests as well as reads. Accordingly, it requires more controls and functions with diverse soft error characteristics. Figure 2.9 is the SES chart for the D-cache. The SES levels of the D-cache are higher for *Bubble* and *Queens* than other programs because their operations are memory-intensive. However, all SES increases appear only in those sub-blocks that are error-sensitive for the other programs. More FILs seem to be sensitive than in the I-cache. The SES level of the D-cache is directly proportional to its access frequency.

FIL 53 marks the validity of data in the bus. An erroneous transition from *valid* to *invalid* simply makes the processor wait for valid data, whereas a transition in the opposite direction lets unexpected data to be delivered to the IU or SMU. A corrupted data transfer also occurs if FIL 54 or FIL 83 fails. FIL 54 aligns data items read from the DCRAM (FIL 83) and puts them on the bus. Since the functions of FIL 54 and FIL 83 are performed in a sequence, their SES level and pattern are very similar. FIL 56 is the most sensitive sub-block as it signals synchronization error, I/O error, and/or memory error in data access. Like FIL 38 in the I-cache, FIL 59 may modify data in the DCRAM with a false write enable signal. Other write

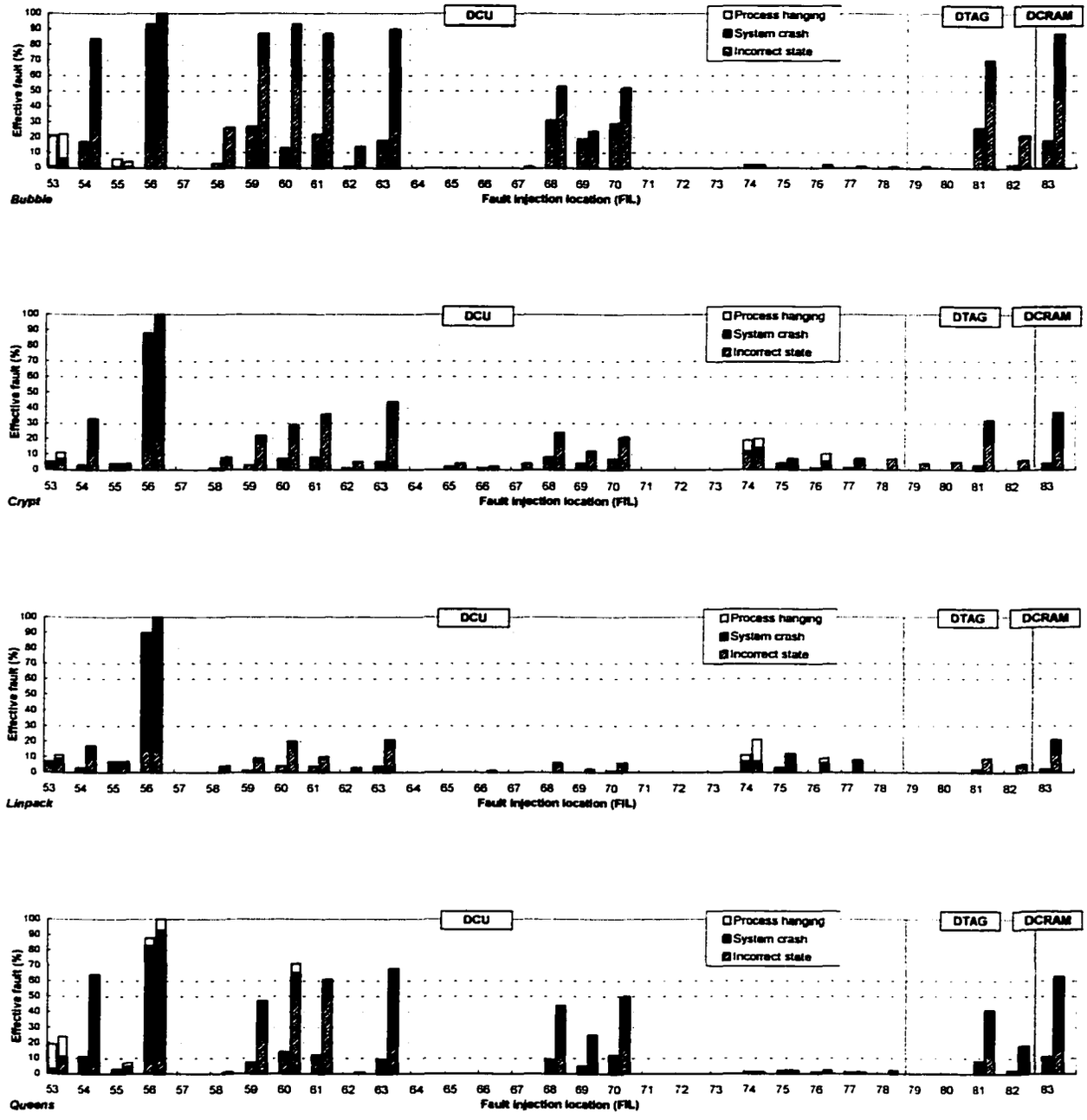


Figure 2.9 The SES of the D-cache.

enable signals are produced by FIL 68 for tags and FIL 69 for status bits. FIL 60 acts as a *sleep* signal to the DCRAM and DTAG. Improperly disabling the D-cache results in a process failure. FILs 61 and 70 are the address for the D-cache access. An address error may be resolved as an access miss or other situations as we discussed with FIL 1. Thus, their sensitivities are close to that of FIL 1. FIL 63 controls bypassing data path for non-cacheable loads and stores. If it selects a wrong path at an active cycle, a memory transaction error occurs.

A DTAG error creates an incorrect hit or miss decision. Miss rate on the D-cache is relatively high, and therefore, more false hits (erroneous decision changes from miss to hit) may occur than in the I-cache. Moreover, a false miss for a dirty line makes a stale data delivery from the memory. This is why the DTAG exhibits some error susceptibilities unlike the ITAG. A bit-toggle at the outputs of FIL 81 or 82 guarantees a mis-decision. On the other hand, faulty FIL 79 still has a possibility of avoiding false hits because the output signal is 19 bits. We need to note that how often a fault in a FIL eventually corrupts its output signal is a different issue from how the output error affects the computation. The former is mainly related with the upset rate of the FIL and the latter is the SES that we investigate in this dissertation. In reality, the upset rates for FIL 79 and FIL 81 might be similar.

When FID was increased to 500 for testing rarely sensitive sub-blocks, faults were effective in FILs 64, 71, and 72, but not in FILs 57 and 73 as shown in Figure 2.7. The reason for FIL 57 is the same as FIL 44 of the ICU. FIL 73 notifies whether or not a store on a SC write miss is completed in the D-cache. The SC miss is a rare event. A false signaling in one direction induces more wait cycles and the signaling in the other direction is not very prone to a failure.

The general protection scheme for the D-cache can be the same as the I-cache in that memory arrays are checked with an ECC and at least critical control sub-blocks are covered. For a write-back D-cache as employed in picoJava-II, up-to-date data items may be available only in the cache. Therefore, more integrity checking needs to considered for the D-cache than the I-cache. Write through policy is popular when parity protection is used. In the case of error detection in a cache line, simply invalidating the line accomplishes error recovery.

2.5.4 FPU, SMU, PCSU, BIU, and MEMC

In Figure 2.10, the SES results for the remaining FUBs are presented where FID is 1 and 10. Apparently, the more floating point operations, the higher SES. However, the FPU here shows very low sensitivities. The reason for that is as follows. FIL 84 is another example of Figure 2.5b in asserting the start and end signal of floating point operations. The output of

FIL 85 is valid only for one or two cycles on completion of a long execution (up to 2000 cycles) of a floating-point instruction. It is quite unlikely for short FIDs to overlap with such a brief interval. To examine the SES of the FPU in the worst situation, we could have injected the faults when the output is active. In that case, the SES would be much higher. Separate fault injections into smaller logic blocks of FIL 85 are needed to understand error manifestation behavior at its output.

The SMU shows zero SES in all FILs except FIL 92 for *Queens*. It handles spills and fills of the SC and manages execution pipeline in overflow and underflow conditions. An overflow occurs if the top of the stack pointer is smaller than the bottom of stack pointer. In comparison to other operations, the overflow frequency is very low. An underflow can be activated only in response to changes in the top of the stack pointer initiated by the return instructions, which account for 0.005% (*Bubble*), 0.001% (*Crypt*), 0.308% (*Linpack*), and 0.573% (*Queens*) of total instructions executed in each program execution. For these reasons, faults are hardly effective in the SMU. FIPs that are randomly selected in our test did not coincide with the occasional SMU active cycles.

Even if FID is 500 (Figure 2.7), FILs 90, 91, 94, and 96 have no changes in the SES. When FID 90 asserts a write enable, the output of FIL 89 is latched. Since the state of FIL 89 does not vary often, additional writes by erroneous FID 90 have usually no special impact. FIL 91 generates pipeline stalls. Holding the pipeline for extra cycles introduces an operation delay, but not a process failure. FIL 94 requests loads solely on a SC fill or an underflow. FIL 96 tells the DCU that a data request is a non-allocate store. This has no effect in the case of a D-cache hit, but on a miss, data item is directly written to the memory. Erroneous switching by FIL 96 between caching or non-caching data cannot alter the correctness of computation unless it caches data from a non-cacheable address region. Based on the observation, the SMU can be the last FUB to consider for integrity checking. Since the SMU mainly consists of random logic, the lifespan of errors in the SMU caused by temporary hardware failures is short.

FIL 97 requests the IU to take a trap when there is an interrupt by an external device. The trap is however not taken when the interrupt mask bit of the processor state register is 0, making faults ineffective. FIL 99 signals a power-down to other FUBs. Unless the IU executes a power-down instruction, indicated by FIL 24, no response is made. Accordingly, its SES is zero. Erroneously triggering a standby when the FUBs are not ready leads the process to fail, whereas not going into the standby mode even at a right timing does not trouble normal execution. Considering this unique characteristic, the power-down control logic needs

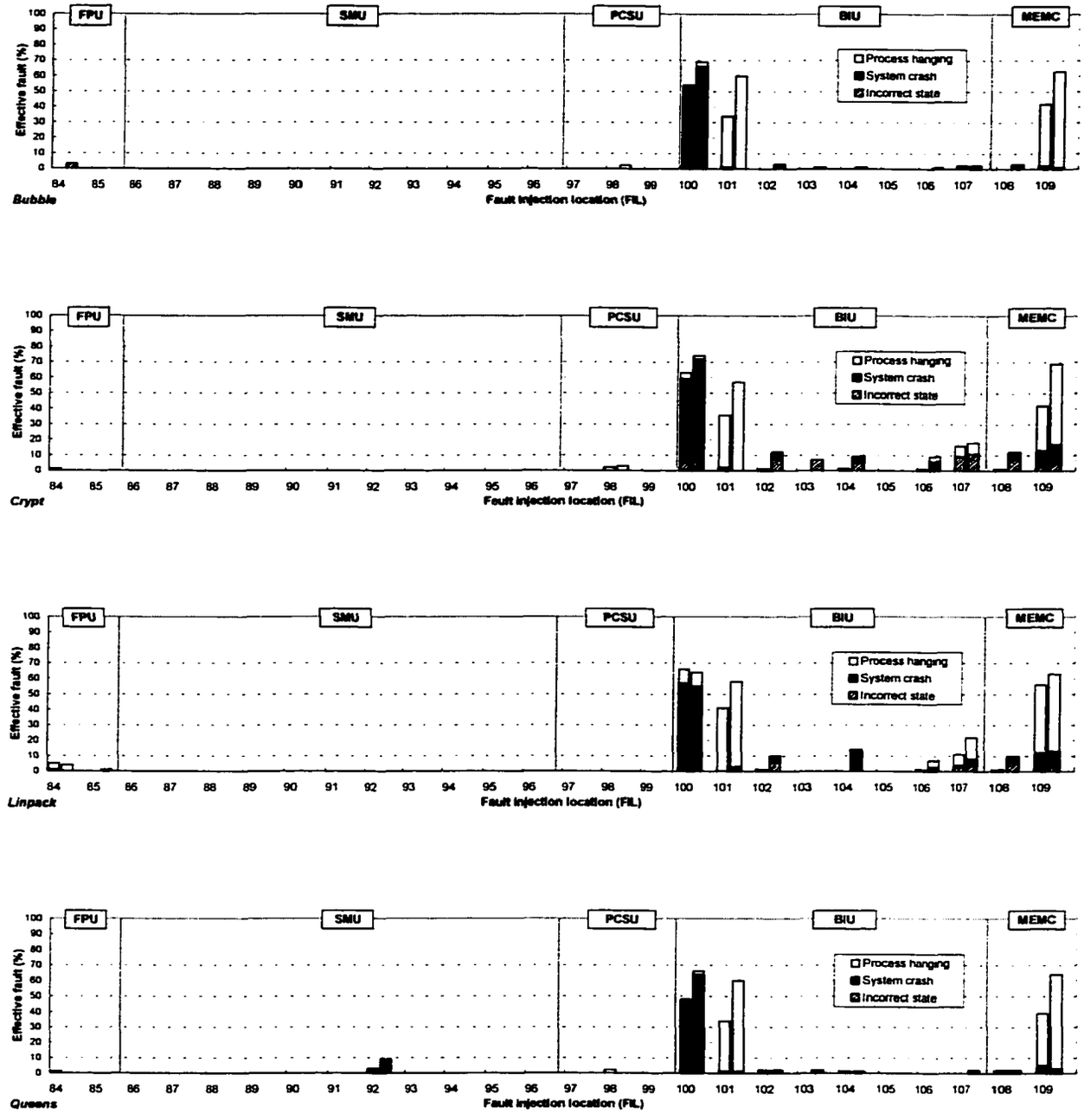


Figure 2.10 The SES of the remaining FUBs.

a protection from the former case only. The PCSU is placed in a fault tolerant position like Figure 2.5b.

As clearly seen in the figure, handshaking signals by FIL 100, 101, and 109 are error-sensitive during memory transactions between the BIU and the caches or the BIU and MEMC. A false bus acknowledgment causes the processor to hang or crash. Other FILs except FIL 107 are less sensitive because their faulty signals neither initiate nor terminate bus transactions. Data errors directly propagate through memory transactions. Thus, ensuring the integrity of handshaking activity along with data check is a critical component, qualifying for protection redundancy.

2.5.5 FUB-level SES

Figure 2.11 visualizes our SES estimate of the picoJava-II processor. The average SES of each FUB is quantized into one of 10 shade levels after weighting the areas and sensitivities of FILs in the FUB. A higher SES is represented by a darker shade.

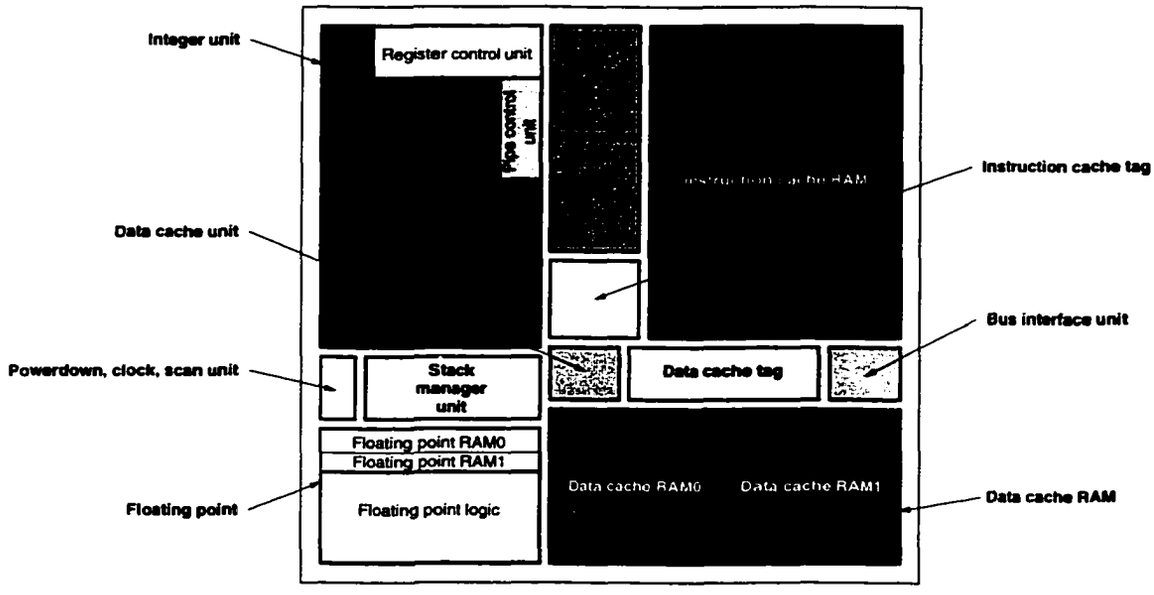


Figure 2.11 The functional unit blocks of the picoJava-II.

2.6 Summary

To cope with increasing demand for dependability enhancement in today's processors, understanding its behavior under faulty environments is a fundamental step. We have conducted fault injection simulations on a commercial product, picoJava-II, to characterize the soft error sensitivities of its components. We summarize our analysis as follows.

- Many processor components are seldom susceptible to soft errors, while there are a few components with a very high SES. The SES of a component is determined by its architectural function; logical situation, governing processor behavior directly or in collaboration with other components; and active cycle/frequency. A logic block may have an inherent capability of fault masking to some degree.
- Workload variation does not convert a critical component to a non-critical component in terms of SES, or vice versa. Although minor changes in the SES level and the pattern of failures can be induced by different workloads, clear distinction between *first-protect* and *last-protect* components is constant. As a result, the SES is an effective standard for prioritizing the integrity checking of various processor components to enhance dependability.

- Soft errors in control logic generally have a shorter lifetime than those in the memory arrays. Controlling program flow and bus transaction is more error-sensitive than other controls. Therefore, protection can start with memory, which is already common in many microprocessors, and then include critical random logic.
- The sensitivities of many components are fairly predictable from processor architecture and organization. The amount of efforts to estimate the SES data is adjustable with varying degree of accuracy. Different levels in processor model hierarchy can be selected for the estimation. This enables the designer to meet time-to-market requirements, while achieving a resource-efficient increase in processor dependability.

Fault injection has been commonly used to judge their effectiveness after fault tolerance techniques are implemented into a processor. Our study demonstrates that fault injection is also a suitable method to improve the protection efficiency of the techniques in an early development stage.

CHAPTER 3 Integrity Checking Architectures for Memory Arrays

Information integrity in cache memories is a fundamental requirement for dependable computing. Conventional architectures for enhancing cache reliability using check codes make it difficult to trade between the level of data integrity and the chip area requirement. We focus on transient fault tolerance in primary cache memories and develop new architectural solutions to maximize fault coverage when the budgeted silicon area is not sufficient for the conventional configuration of an error checking code. The underlying idea is to exploit the corollary of reference locality in the organization and management of the code. A higher protection priority is dynamically assigned to the portions of the cache that are more error-prone and have a higher probability of access. The error-prone likelihood prediction is based on the access frequency. We evaluate the effectiveness of the proposed schemes using a trace-driven simulation combined with software error injection using four different fault manifestation models. From the simulation results, we show that for most benchmarks the proposed architectures are effective and area efficient for increasing the cache integrity under all four models.

3.1 Conventional Memory Protection

Memory hierarchy is one of the most important elements in modern computer systems. The reliability of the memory significantly affects the overall system dependability. The purposes of integrating an error checking scheme in the memory system are to prevent any error that has occurred in the memory from propagating to other components and to overcome the effects of errors locally, contributing to the overall goal of achieving failure-free computation.

Transient faults can corrupt information in the memory, i.e., instruction and data errors. These soft errors may result in erroneous computation. In particular, errors in cache memory, which is the closest data storage to the CPU, can easily propagate into the processor registers and other memory elements, and eventually cause computation failures. Although the cache memory quality has improved tremendously due to advances in VLSI technology, it is not possible to completely avoid transient fault occurrence. As a result, data integrity checking,

i.e., detecting and correcting soft errors, is frequently used in cache memories.

The primary technique for ensuring data integrity is the addition of information redundancy to the original data. Whenever a data item is written into the cache, a check (or protection) code such as parity or error-correcting code (ECC) is also included. We denote a pair of data and check code by a *parity group*. When an item is requested, the corresponding parity group is read and an error syndrome is generated to check and correct the error if there is one. The capability of the protection code needs to be determined properly depending on the degree of required data integrity, expected error rate based on harshness of the operating environment, and design and test cost.

Despite the fact that predicting the exact rate and behavior of transient faults in a system is not possible, current data integrity checking schemes for caches are generally selected on a single-bit failure model basis. Thus, byte-parity scheme (one bit parity per 8-bit data) [80] and single error correcting-double error detecting (SEC-DED) code [71] are widespread. Many higher capability codes for byte or burst error control have also been studied [8], [74].

Check codes employed for increased reliability in the caches are constructed in a *uniform structure*, i.e., every unit of data is protected by a check code of the chosen capability. This conventional method is reasonable under the assumption that each cache item has the same probability of error occurrence. However, it has the following deficiencies.

- Check code in the uniform structure is an expensive way to enhance cache reliability. Therefore, it is overkill under extremely low error rates.
- It is not flexible in terms of chip area requirement, as the area occupied by the check code is directly proportional to the cache size. If the budgeted area is not sufficient for the uniform structure, no intermediate architectures are currently available. The high overhead may result in sacrificing the integrity checking.

The uniform structure enables every item to be checked. However, error checking is necessary only for those items that are likely to be corrupted. If it is possible to predict such cache items, a higher data integrity can be achieved with a smaller amount of chip area for the check code. In practice, there are several reasons that soft error occurrence tends to concentrate in a few locations. Information in the cache can be altered during read/write operations due to low noise margins, and thus cache lines that are frequently accessed may have a higher probability of corruption. Cross-coupling effects may also induce errors in neighboring locations of a line being accessed. On the other hand, global random disturbances commonly affect any location. More importantly, errors in unused lines are no concern.

In this chapter, we take these factors into account to develop area efficient architectural solutions for improving cache integrity. The underlying idea is that more error-prone and more likely used cache lines must be protected first. The random faults are not biased to a specific location or time. However, if a fault occurs during the access of a line, it is more likely to affect the data being accessed. As a result, access frequency makes a difference in the probability of error occurrence between *active* (more access) and *inactive* (less access) lines. With large caches, the majority of cache accesses are usually localized in a small portion of the cache. This frequently accessed part is considered more error-prone. The corrupted items in the most frequently used (MFU) lines are likely to be used as instructions or operands, quickly affecting the computation. On the other hand, errors in inactive lines have a higher probability of being replaced or overwritten with new, correct data [79]. Data errors are harmful only if they are used for operation, suggesting that not providing check codes for inactive lines may not affect the integrity of the computation.

We present three new architectures, *parity caching*, *shadow checking*, and *selective checking*, to protect primary caches. These schemes allow flexible trade-offs between silicon area and level of data integrity so that both the reliability and area requirements can be met. The new schemes can achieve an acceptably high level of fault coverage with much less area than the uniform structure, realizing area efficient enhancement of cache system reliability.

3.2 Errors in Cache and Their Effects

To reduce CPU-memory bandwidth gap, up to 60% of the area of recent microprocessors is dedicated to caches and other memory latency-hiding hardwares [75]. The cache memory stores instructions or data in data RAM along with address tags in tag RAM. The primary caches are required to operate at the processor's clock speed. Use of lower voltage levels, high speed data read/write operations, and extremely dense circuitry increase the probability of transient fault occurrence, resulting in more bit errors in cache memories. Moreover, external disturbances such as noise, power jitter, local heat densities, and ionization due to alpha-particle hits [76] can also corrupt the information.

Figure 3.1 shows how the state of a cache data block is affected by error occurrence and recovery. Initially, the block is error-free (State N). Single- and multiple-bit errors due to some transient faults lead to State S and State M, respectively. The state of a corrupted block changes back to State N if the error is overwritten by a new correct item or is corrected

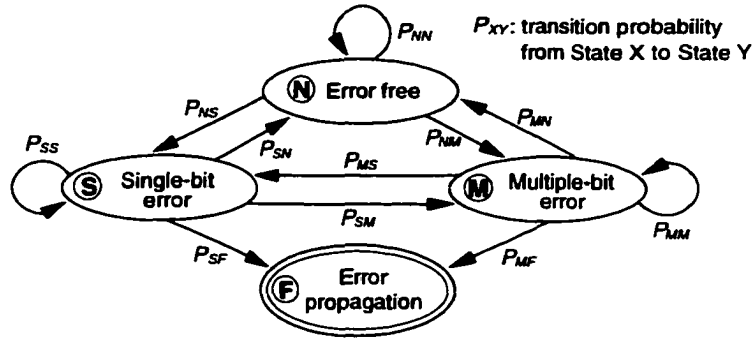


Figure 3.1 Cache data state transition.

by a recovery mechanism. The absorbing state, F, represents the situation where an error propagates outside the cache boundary through a normal access. If the cache memory always operates in State N ($P_{NN} = 1$) or erroneous items are never used, then no fault tolerance schemes are necessary. However, in practice, this is not likely to be the case. The check codes help keep P_{SN} and P_{MN} nearly equal to 1 so that the cache block rarely reaches State F (i.e., $P_{SF} \simeq P_{MF} \simeq 0$).

One may suspect that extremely infrequent error propagation ($0 \neq P_{SF} << 1, 0 \neq P_{MF} << 1$) may not have any notable effects. However, even a single-bit error can bring a complete system failure. Through program execution, corrupted data items propagate to the processor's registers and produce an erroneous outcome that can eventually propagate to the external world. A single error can also spread to other registers, cache lines, and memory locations as the processor continues to use the corrupted data recursively. The erroneous contents of registers can also cause page or segmentation faults and incorrect control flow changes. It is shown in [79] that the probability of a single bit leading to a failure is about 50%. However, the actual probability heavily depends on the cache hit rate.

Bit changes in the tag RAM also cause the following improper cache hit and miss decisions that make the processor's memory references chaotic.

- Pseudo-hit: the tag portion of the incoming reference address matches with the wrong cache line's tag field.
- Pseudo-miss: the tag associated with the desired data item does not match with the reference address.
- Multi-hit: the tag portion of the reference address matches with the tags of multiple lines in a set.

In the case of a pseudo-hit, the processor gets wrong data on a read and updates the data in the wrong location on a write. A pseudo-miss generates an unnecessary main memory access. The multi-hit may be detected by the cache controller without check code support for the tags, but handling is not simple. The controller cannot distinguish between the multiple hit lines to service the processor's request. Moreover, invalidating all of those lines and treating the access as a miss is not a solution if any of the lines are dirty, i.e., valid data may exist only in the cache. Writing the data back to main memory should precede the invalidation of a dirty line. However, this cannot be done without resolving the line selection problem. Thus, a resolution may not be possible or may lead to data consistency failure.

Due to an error in the cache status field, a valid line can be unintentionally invalidated if its valid bit is changed erroneously. In the case of a dirty bit error, a dirty line is considered to be clean and may be replaced without a write-back. Therefore, the most recent data can be lost. Line replacement based on access history can also be performed improperly if faults flip the corresponding history bits.

3.3 New Architectural Approaches

In conventional systems, data integrity checking schemes are implemented in a uniform structure. In this section, we describe three alternative architectures that have flexible chip area requirements. The parity caching scheme described in section 3.3.1 is proposed as a substitute for uniformly organized check code under extremely low error rates. section 3.3.2 describes shadow checking, which is an inexpensive variant of replication architecture under very noisy environments. Selective checking is presented in section 3.3.3 as a simpler alternative to the first two when a cache has multi-way set associativity. In section 3.3.4, we also discuss integration of cache scrubbing [77] into the proposed architectures to enhance their capabilities.

A data read/write involves accessing cache cell arrays including data, tag, and status bits. Errors can appear in any field. Therefore, integrity checking is required for all three fields. For brevity in presentation, we do not always address the proposed schemes separately for each field. However, the operation and management mechanisms apply to all three fields in an identical manner.

3.3.1 Parity caching

One of the widely known program properties is that only 10% of program instructions are responsible for 90% of instructions executed [70]. For some programs, a similar observation can be made in the data segment of main memory. Cache accesses are also often localized. Under considerably low error environments, it can be expected that most soft errors of any significance will occur in these most commonly used portions of instructions and data.

In a low error rate environment, when the budgeted area is not sufficient for the check codes of the uniform structure, the number of check codes needs not be continuously increased with the primary cache size to maintain high data integrity. Based on the assumption that the MFU lines are most error-prone and errors in those lines easily propagate unless checked, we organize a *parity cache*, whose entries contain the check codes for the MFU lines. This scheme is called *parity caching*: the caching of check codes.

The organization and operation of the parity cache are similar to general cache memories, but it provides integrity checking for the main cache. It covers the most error-prone main cache locations using $log_2 l$ index bits, where l is the number of lines in the main cache. The number of parity cache entries, n, is smaller than l. The main cache lines for which check codes are held in the parity cache are selected dynamically such that the MFU portion of the cache can be protected first. This is accomplished by employing least recently used (LRU) replacement policy for the parity cache, where the entry that has not been used for the longest time is replaced with a new item.

Figure 3.2 shows the logical organization of a 16KB direct-mapped data cache or D-cache (left half) protected by a parity cache of 16 entries (right half) in conjunction with an ECC unit. The 16 parity cache entries are organized in a 4-way set associative manner and store check codes for 16 lines selected from the main cache. The ECC unit performs error checking. The parity cache type and the check code capability can be flexibly determined. The main data line consists of 32 bytes, and 32 parity bits (1 per byte) are used for its protection. The tag is protected with a SEC-DEC code. For the status bits, one even parity bit is used.

For the mapping between the parity and main caches, each entry in the parity cache is tagged. In the case of direct-mapped main caches, the index field of a reference address is used for the parity cache tag as it exactly corresponds to one line in the main cache. In the example above, the first seven bits of the index are stored as a tag and the last two bits are used in selecting a set in the parity cache. Note that the number of parity tag bits is small in comparison to the main cache tag, resulting in simpler tag comparators. If the parity

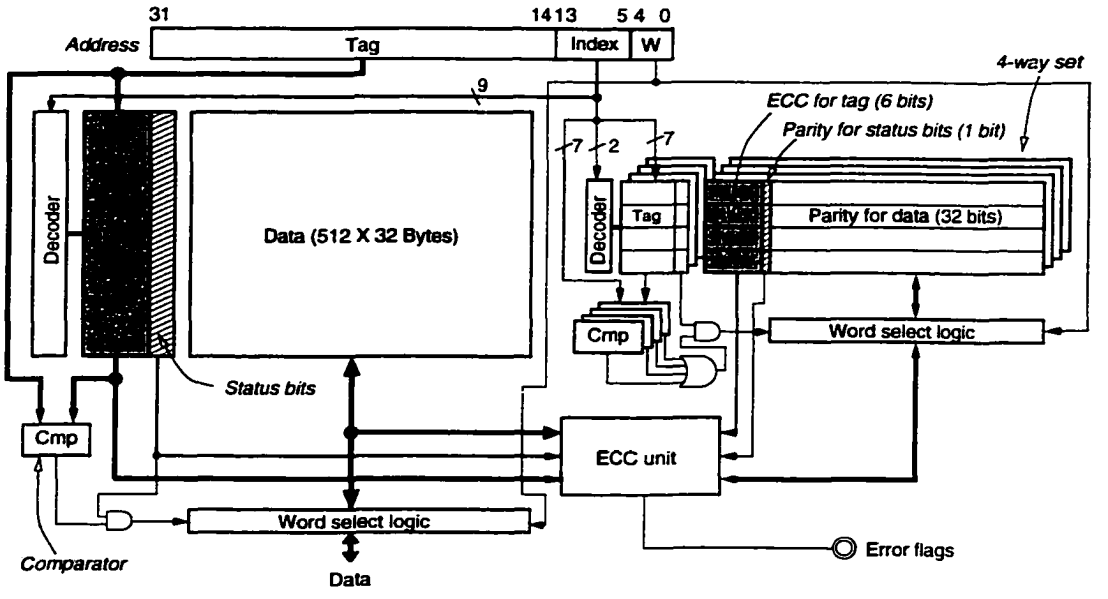
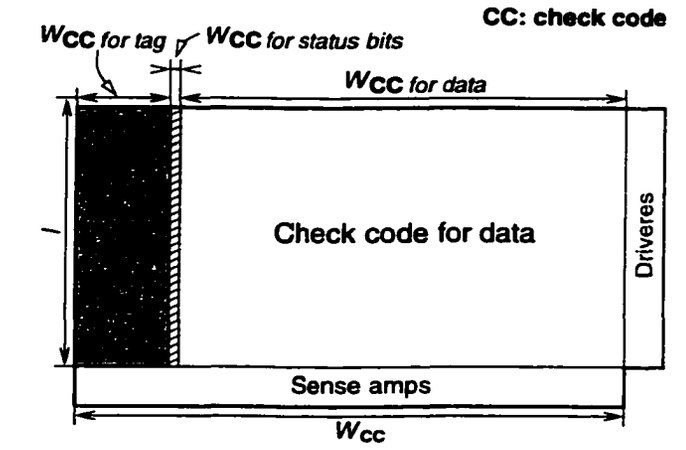


Figure 3.2 A 16KB D-cache and a parity cache.

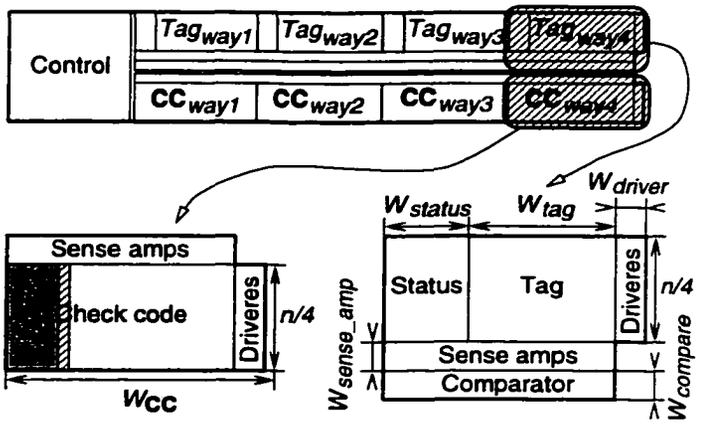
cache uses direct-mapping, the tag is reduced to five bits. If the main cache is k-way ($k > 1$) set associative, multiple lines of the same index value can coexist in the main cache and one parity entry may erroneously be mapped to all of those lines. This problem is solved by storing additional $log_2 k$ bits in the parity tag to distinguish the correct line from k ways. In this section, we apply the parity caching only to direct-mapped main caches. We present an alternate method (selective checking) for set associative main caches in section 3.3.3.

While the main cache serves the processor's request, the parity cache synchronously monitors the integrity of the main cache and updates the check codes. This is the same as existing systems with the uniform array of check codes, and so, no additional delay in cache access time is needed. When a miss occurs in the main cache, new data items are fetched from the lower level memory with check codes, and they are written into the parity cache in a free entry, if any, or replace the LRU entry of the mapped set. Whenever a read hit occurs in both caches, error checking on all three fields of the line is performed using the ECC unit. Handling errors is the same as in the conventional systems. Whenever a main cache line is replaced, the check codes in the corresponding parity entry are also updated for the new line. Thereby, a hit in only the parity cache never occurs.

If the parity cache misses the entry for a data block being requested by the processor, the integrity checking cannot be processed. In this case, the check codes are computed from the data in the main cache and stored in a selected parity entry for future protection. If



(a) Conventional array



(b) 4-way parity cache

Figure 3.3 Check code area model.

the accessed line has a corrupted item, error propagation is possible. Check codes generated using erroneous data do not help error checking. This event is denoted by *misconstruction*. Although the error does not always propagate, we assume the worst case and treat it as error propagation in the evaluation of the scheme.

The area estimate of the parity cache is obtained from an on-chip cache area model presented by Mulder et al. in [73]. They have used the technology independent notion of a register-bit equivalent or *rbe*. One *rbe* equals the area of a one-bit register storage cell that has the highest bandwidth. The static storage cell of medium bandwidth that we use here has been empirically determined to be 0.6 *rbe*. Thereby, an area represented in *rbe* for register cells is converted to static cell area by multiplying by a factor of 0.6.

Figure 3.3 depicts the check code array of a conventional cache in the uniform structure

and a k-way set associative parity cache ($k = 4$). We denote the total area of the two models by $\mathcal{S}_c$ and $\mathcal{S}_p$, respectively. Let W_{E} be the width of an element E and let CC represent the check code. The area is the sum of areas of all memory elements and is given by

$$\begin{aligned}\mathcal{S}_c &= \mathcal{A}(\text{CC for tag}) + \mathcal{A}(\text{CC for status bits}) + \mathcal{A}(\text{CC for data}) \\ &\quad + \mathcal{A}(\text{drivers}) + \mathcal{A}(\text{bitline sense amplifiers}), \qquad (3.1)\\ \mathcal{S}_p &= \mathcal{A}(\mathsf{CC} + ovhd_{\mathsf{CC}}) + \mathcal{A}(\text{tag} + ovhd_{tag} + \text{status} \\ &\quad + ovhd_{status}) + \mathcal{A}(control), \qquad (3.2)\end{aligned}$$

where $\mathcal{A}(\mathsf{M})$ and $ovhd_{\mathsf{E}}$ denote the area of module M and the overhead for element E, respectively. The $ovhd_{\mathsf{E}}$ includes comparators if any, drivers, and sense amps. In the implementation of MIPS-X [68], $W_{compare}$, W_{driver}, and $W_{sense\text{-}amp}$ are approximately 6 *rbe* each. For a static cell array of $l \times W_{\mathsf{CC}}$ bits (Figure 3.3a), the total size (Eqn. (3.1)) is

$$\begin{aligned}\mathcal{S}_c &= 0.6(W_{\mathsf{CC}} + W_{driver})(l + W_{sense\text{-}amp}) \\ &= 0.6(W_{\mathsf{CC}} + 6)(l + 6),\end{aligned}$$

where $W_{\mathsf{CC}} = W_{\mathsf{CC}\ for\ tag} + W_{\mathsf{CC}\ for\ status\ bits} + W_{\mathsf{CC}\ for\ data}$.

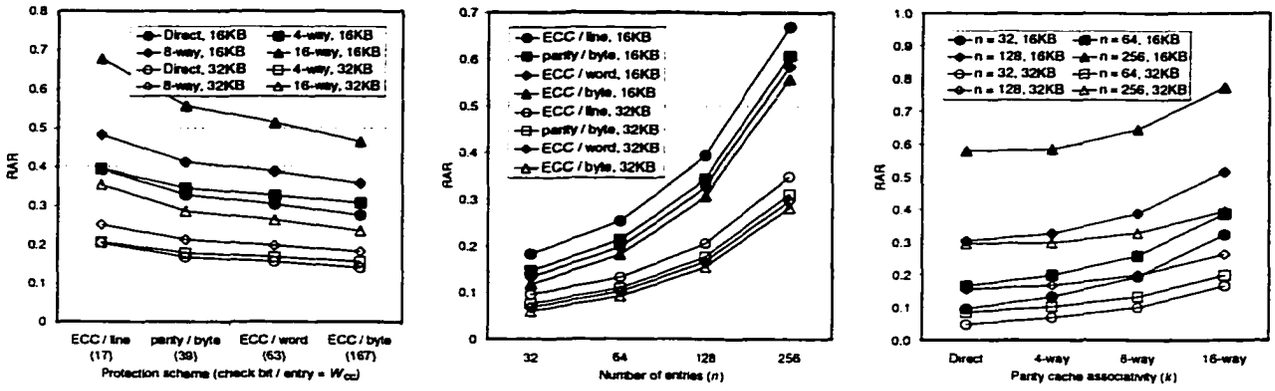
The control logic for the parity cache can be implemented in a programmable logic array (PLA) as a part of the main cache controller. A PLA of 130 *rbe* is presumed according to [73]. From Eqn. (3.2), the area of a k-way parity cache of n entries for $k \neq n$ (Figure 3.3b) is obtained as

$$\begin{aligned}\mathcal{S}_p &= 0.6(W_{\mathsf{CC}} \cdot k + 6)(\tfrac{n}{k} + 6) + 0.6\{(W_{status} + W_{tag}) \cdot k \\ &\quad + 6\} \cdot (\tfrac{n}{k} + 6 + 6) + 130 \\ &= 0.6n\left(\Delta_{\mathsf{CC}} \cdot W_{\mathsf{CC}} + \Delta_{tag}(W_{status} + W_{tag})\right) + 195, \qquad (3.3)\end{aligned}$$

where W_{status} = LRU bits + valid bit + parity for the tag = $\log_2 k + 1 + 1$, $W_{tag} = \log_2 l - \log_2 n + \log_2 k = \log_2 \frac{l \cdot k}{n}$, $\Delta_{\mathsf{CC}} = 1 + \frac{6 \cdot k}{n} + \frac{6}{W_{\mathsf{CC}} \cdot k}$, and $\Delta_{tag} = 1 + \frac{12 \cdot k}{n} + \frac{6}{(W_{tag} + W_{status}) \cdot k}$.

To compare the areas of the two organizations, we compute relative area ratio[1] (RAR). The RAR for parity caching equals $\frac{\mathcal{S}_p}{\mathcal{S}_c}$. Figure 3.4 plots the RARs for various sets of configuration parameters: check code type, data unit size, number of parity entry, and associativity. Considering current microprocessors, two main cache sizes, 16KB for instruction cache (I-cache) and 32KB for D-cache, are compared. However, some microprocessors employ larger caches. In that case, the RARs become even better for parity caching. The number of check bits

[1] We also use this metric for shadow checking and selective checking.



(a) $n = 128$ (b) $k = 4$ (c) $W_{CC} = 63$

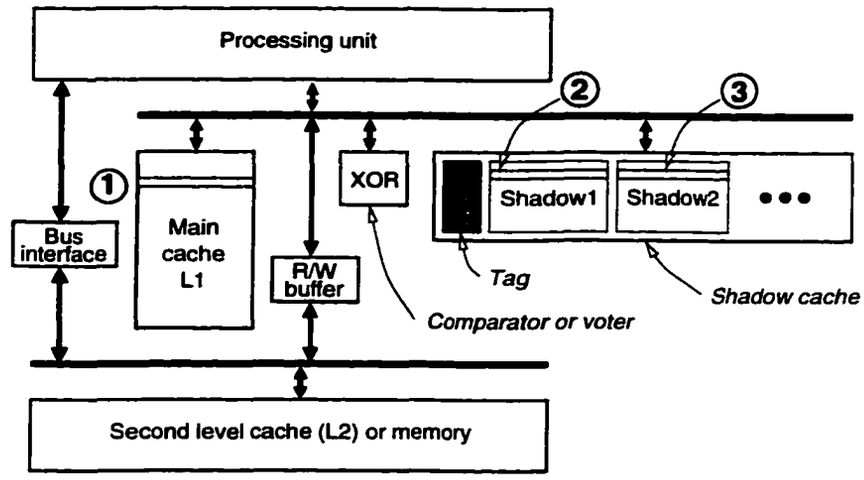
Figure 3.4 Relative area requirement.

per entry corresponds to W_{CC}. Four values are compared, representing different protection capabilities. Several conclusions can be drawn. An increase in check code width results in a decrease in the RAR. Obviously, more overhead is required for higher associativity and the RAR is proportional to the number of entries. The parity cache with an RAR of less than 1 is of interest to us. The corresponding protection coverages for these organizations are given in section 3.5.

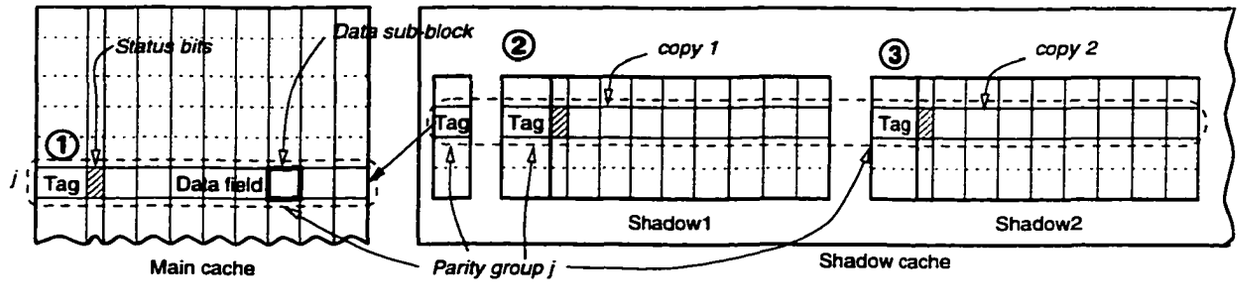
3.3.2 Shadow checking

For applications that require very high data integrity or operate under highly noisy environments, parity- and ECC-based protection cannot be satisfactory. One general approach in this case is to use replicated architecture such as N modular redundancy (NMR) with majority voting, but it is very expensive. Instead of full replications, we present an alternative approach, called *shadow checking*, where multiple copies are partially supported to meet a budgeted area which is not adequate for a complete NMR. The copies of the MFU lines are stored in *shadow cache*. The underlying idea is the same as parity caching, but the shadow cache performs error checking by means of comparison using the copies of data rather than check codes. The goal is to obtain a high reliability enhancement even in the presence of multiple-bit errors with smaller chip area overhead.

Figure 3.5a shows the diagram of shadow checking architecture. Depending on space availability, N identical additional cache modules, called *shadow* i for $1 \leq i \leq N$, are included in the shadow cache. We adopt the same address mapping mechanism used in parity caching. Figure 3.5b depicts a parity group, j, consisting of a shadow cache tag and N copies of infor-



(a) Basic organization



(b) Components of parity group

Figure 3.5 Shadow checking architecture.

mation, each of which contains tag, status, and data bits. The shadow cache operates like a shadow of the main cache. Data written into the main cache is also written into the shadow cache along with the corresponding tag and status bits in parallel. Error checking is performed on read hits in both caches, and its effect depends on the number of erroneous *shadow* modules and their error patterns. This is equivalent to known reliability gain in an NMR system [78]. Thus, we do not discuss it here.

We show the advantages of shadow checking over full replication with respect to chip area requirement and resultant reliability enhancement. In the comparison, we ignore the common factors of the two architectures such as comparator/voter reliability and delay, and synchronization cost. Figure 3.6 shows the RARs of a shadow cache of two shadow modules in comparison with a triple modular redundant (TMR) cache using the same area model presented in section 3.3.1. The RARs are smaller than those of a parity cache for the same parameters due to the higher overhead of a conventional TMR system.

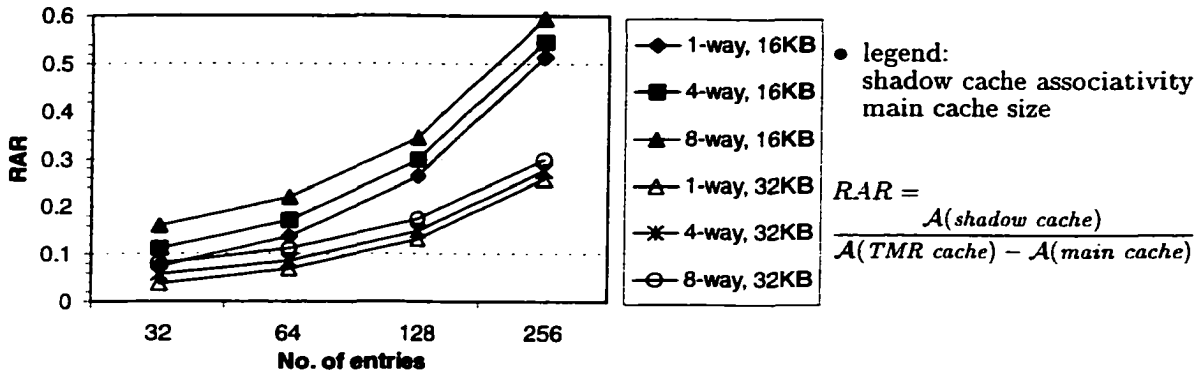


Figure 3.6 Relative area ratio.

3.3.3 Selective checking

Parity caching and shadow checking have been proposed as alternative architectures to the uniform check code structure and replication method, respectively. If the main cache has k-way ($k \geq 2$) set associativity, we can also configure redundancy in a simpler manner. Out of k lines per set, only s lines ($1 \leq s < k$) are selected to assign the check codes. Similar to the previous two schemes, line selection is based on the access frequency. We simply choose the most recently used (MRU) lines of each set for error checking with the expectation that those lines are MFU, and thus error-prone and likely to be accessed in near future. We call this scheme *selective checking*. It is obvious that the RAR for selective checking is approximately $\frac{s}{k}$.

Figure 3.7 depicts a comparison of redundant code organizations between conventional approaches (left column) and selective checking (right column) for $k = 4$ and $s = 1$. Many commercial microprocessors use byte-parity or SEC-DED codes in the uniform structure as shown in Figure 3.7a. Alternatively, in selective checking with $s = 1$ (Figure 3.7b), for Set i, only the MRU line is protected by a check code. (If $s \geq 2$, s MRU lines are guarded.) Each check code entry is independently assigned to any line of a set while keeping track of the MRU. Similar concepts can be applied to the NMR cache (Figure 3.7c) for a selective NMR (Figure 3.7d) where N copies are maintained for only the MRU portions. In the case of a miss, a line is fetched from memory with check code and it becomes the MRU line. Whenever the MRU line is requested, the cache controller recognizes that the current check code belongs to the MRU and performs error correction. As a result, no tag and resultant overhead are necessary to achieve dynamic mapping.

The reliability of a cache that is already equipped with a check code can be further enhanced

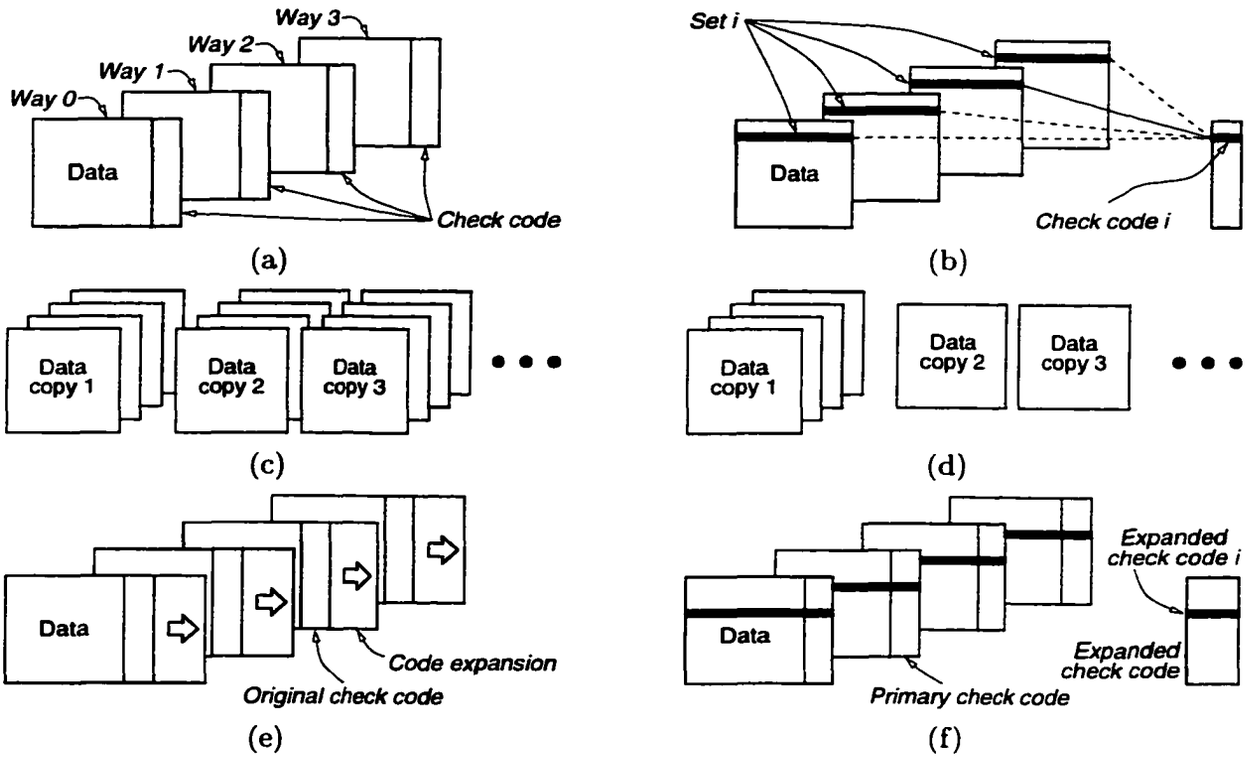


Figure 3.7 Uniform vs. selective organization.

in many ways. One approach is to expand the check code with more check bits as shown in Figure 3.7e. The new wider check code has a higher checking capability. In case the full expansion is not affordable, we can adopt the selective structure here. Only s additional code entries for each set are provided to enhance the protection of the MRU lines. Figure 3.7f shows the selective expansion for $s = 1$. The combination of primary and expanded check codes is called *enhanced check code*. The primary check code is separable from the enhanced check code. The expanded code is built in such a way that the chosen line is protected by more intensive checking in conjunction with primary check code. Any non-MRU line of a set turns into a new MRU line whenever it is accessed. In this case, only the checking by the primary code is valid. The expanded code is ignored and is replaced by the code for a new MRU line. Other basic operations of enhanced check codes are the same as for the previous cases.

In selective checking, the redundant code entry is constructed for only s ($< k$) lines per set. Thus, the redundant codes are always evenly distributed over different sets irrespective of their usage frequency. On the other hand, the parity and shadow caches maintain the redundant code entries for the MFU lines in the range of the entire main cache. This results in differences in cost and protection coverage of the selective checking compared to the other two schemes.

3.3.4 Cache scrubbing

For most programs, less than 30% of instructions are memory references. The D-cache is occupied during the executions of those instructions. Depending on the processor architecture, some D-cache cycles may be idle. To further enhance the data integrity, we can scrub off the latent errors in the D-cache whenever possible. Soft error scrubbing is accomplished by reading out the data and check bits, verifying their correctness, and writing back the corrected data [77]. Scrubbing is more advantageous to caches protected by a low capability check code.

Since in our proposed schemes we shrink the check code array, we suggest the use of the cache scrubbing technique to increase the protection coverage. On every idle cycle, the cache controller executes a single scrubbing cycle using an entry from the check code array and its corresponding line in the cache. One question that arises here is how to pick a line for scrubbing. Random selection is the easiest method to implement but performance may be poor. Intuitively, it could be beneficial to check the lines whose check codes are expected to be discarded soon to make room for new codes. These can be the LRU lines in consideration of temporal locality. By also taking spatial locality into account, lines away from the MRU line and their neighboring lines can also be selected.

3.4 Error Model and Evaluation Methodology

For the evaluation of the schemes, we employed a trace-driven simulation combined with software error injection. An error injection process inverts a single or multiple bits in any field of a selected line. To reflect diverse possible fault manifestation patterns, we conducted error injection based on the following four error models.

1. For a cache item access, the mapped line/set in the RAM is activated and probed. Any fault during the access can result in errors in the line being accessed. Thus, a higher error probability is expected in more frequently accessed lines. Under this error model, the error injection is executed in the target line right after the access. We call this *direct injection.*
2. Cross-coupling effects of faults can generate errors in the adjoining lines of the currently accessed line. During a line access, an error is injected in a neighboring line on either side. We call this *adjacent injection.*
3. Independent of line access, external disturbances and single-event upsets can generate soft errors in any location at any time. To simulate this occasion, an error is injected in

a random location at a random time. This is called *random injection.*

4. Unlike previous models, faults can cause errors in a group such as column, row, or cell cluster. Only column errors can induce a performance difference between the conventional and proposed architectures. Thus, we include a model, called *column injection*, where an error is injected in every row of a selected column. This model is added to examine the schemes' performance in the worst situation.

All error models apply to both the main and protection cache or array. The accuracy of error models depends on the nature of the physical faults. Fault behavior and the distribution of different types of faults are likely to vary depending on the operational environment. Therefore, it is very difficult to judge which error model is dominant and realistic in a general situation. For this reason, we carried out a set of simulation experiments for each model separately.

Table 3.1 Base cache parameters.

Parameters	Main cache	Parity/Shadow cache
Size	I-cache: 16KB, D-cache: 32KB	256 check code entries
Associativity	Direct-mapped	4-way set associative
Replacement	Least recently used (LRU) line first	
Write policy	Write-through, write around	
Line size	32 Bytes	32 Bytes (shadow)

The simulations were performed on-the-fly and every operation was handled on a clock-by-clock basis, assuming that a single instruction is issued and finished in every clock cycle on a perfectly pipelined processor. The same protection scheme was applied to both the I-cache and D-cache on each simulation run. Table 3.1 lists the base cache parameters used for the simulations unless specified otherwise. All programs of SPEC95 suite [81] were instrumented on the Sun Ultra1 model using Sun's Shade tool V5.32C [69]. Table 3.2 shows input files and memory access rates of the programs and hit rates on the base caches. These benchmarks provide a range of computation and memory access patterns, and their instruction and data sets are much larger than the simulated cache sizes. All benchmarks' executable files were built using Sun WorkShop Compilers, `cc` and `f77`, with the optimization flags `-fast`, `-xO4`, and `-xdepend`.

The moment at which the decision of an error injection is made is defined as an *injection decision point* (IDP). For direct and adjacent injection models, the end of each read/write access cycle was considered as an IDP, while the trailing edge of every CPU clock cycle was used for random and column injections. At each IDP, an error (multiple errors for the column

injection) is injected with a constant probability, which is set to 10^{-6} for direct and adjacent, 0.2×10^{-6} for random, and 0.5×10^{-8} for column injection. These are accelerated rates for the rare events. If an item selected for the error injection already has an error, no additional error is injected. The number of error injections, I, at N IDPs is a binomial random variable and the error injection rate is I/N. To ignore initial warm up routines, the error injection was started after the first 10 million instructions while the caches operated under normal conditions. Errors were injected independently for the next 500 million instruction executions and no injection was performed afterward. In consideration of latent errors, the simulations were terminated after the following 100 million instructions.

Table 3.2 Summary of benchmarks.

Benchmark	Input file	Load (%)	Store (%)	I-cache (%)	D-cache (%)
compress	bigtest.in	7.20	2.16	99.99	84.68
gcc	cp-decl.i	19.34	5.52	96.40	93.83
go	9stone21.in	18.79	6.77	97.74	93.60
ijpeg	vigo.ppm	17.03	6.60	99.91	89.74
li	*.lsp	20.89	9.89	98.61	93.66
m88ksim	ctl.in	17.17	9.95	97.47	98.33
perl	primes.in	26.03	12.41	96.08	95.76
vortex	vortex.in	18.74	8.47	95.10	91.23
SPECint95		**18.15**	**7.72**	**97.66**	**92.60**
applu	applu.in	25.43	11.40	99.99	86.25
apsi	apsi.in	28.81	12.83	99.76	88.57
fpppp	natoms.in	38.21	11.04	93.66	96.69
hydro2d	hydro2d.in	21.65	9.28	99.52	75.03
mgrid	mgrid.in	38.29	19.84	99.99	95.68
su2cor	su2cor.in	22.24	7.23	97.11	91.52
swim	swim.in	24.34	10.35	99.99	79.19
tomcatv	tomcatv.in	22.20	7.74	97.47	92.57
turb3d	turb3d.in	17.16	12.24	99.90	93.25
wave5	wave5.in	19.33	10.26	97.95	84.01
SPECfp95		**25.77**	**11.22**	**98.53**	**88.28**

The performance comparison targets for the proposed schemes are uniformly organized check code and replication. The number of error bits per injection is not an important factor in the comparison because the same capability of the unit protection code is assumed. Only the number of code entries is different. The parity cache was implemented with a SEC-DED code, and single-bit errors were injected. D-cache scrubbing was tested along with parity caching. A shadow cache of two shadow modules was compared with a general TMR cache. To simulate a harsher environment for shadow checking, multiple-bit errors were used. Although we proposed three organizations for selective checking, the main idea of the three is common. Therefore, we investigated only a simple case where only s entries of SEC-DED codes are maintained for each main cache set (Figure 3.7b) with single-bit error injection.

Our main interest is how many injected errors propagate to other components under the proposed schemes. For a quantitative performance measurement, we use error propagation rate (EPR), defined by

$$EPR = \frac{\textit{total number of errors propagated}}{\textit{total number of errors injected}} \times 100.$$

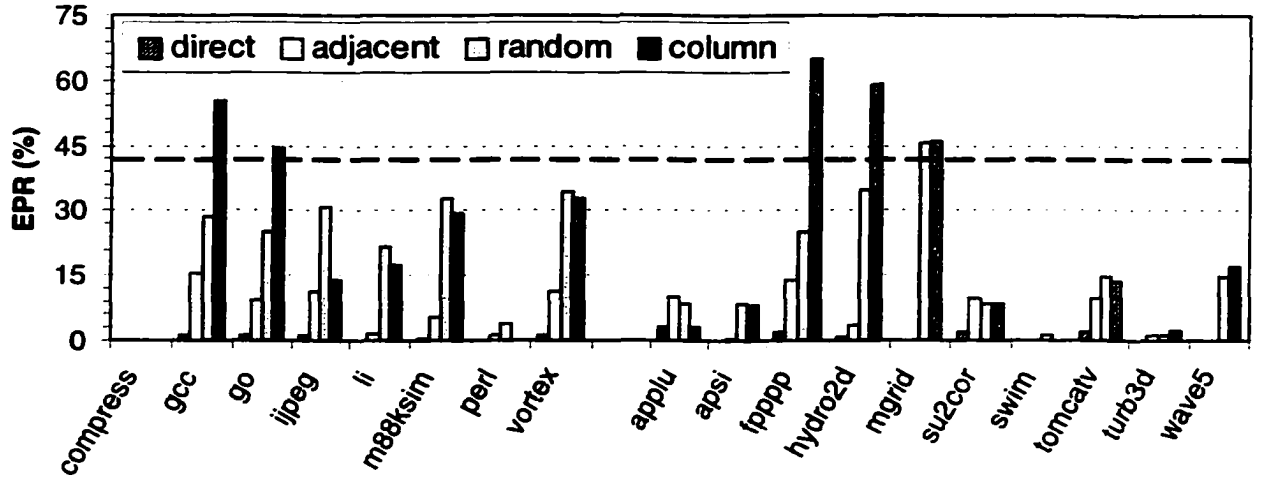
3.5 Results and Analysis

This section reports the performance of the three proposed architectures. Instead of presenting the simulation results in an exhaustive manner for all parameters, we focus only on the salient features to gain insight into how the parameters affect the protection capabilities of the schemes.

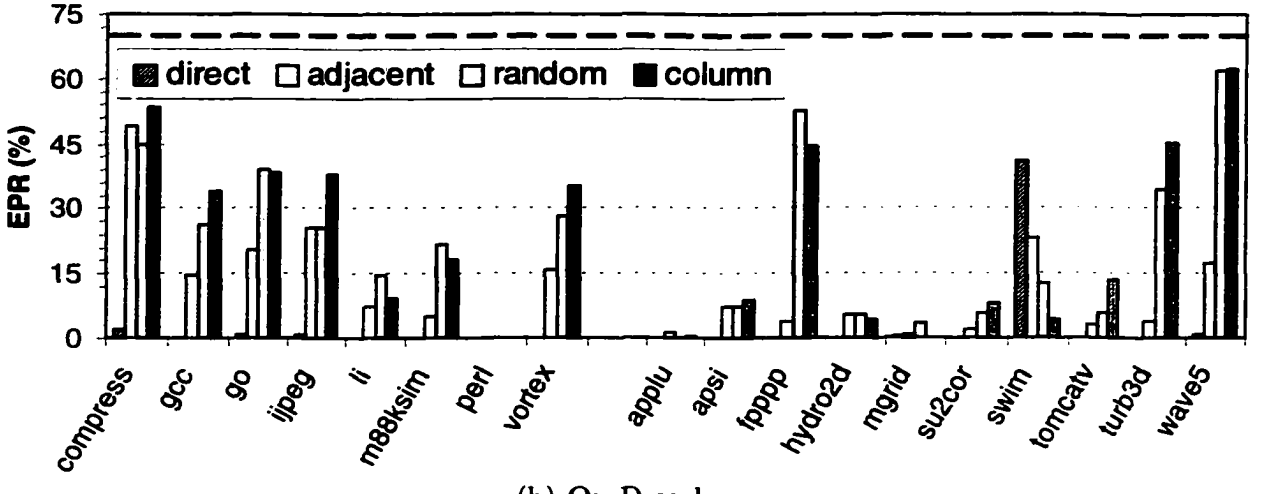
3.5.1 The performance of parity caching

Recall that single-bit errors in cache protected by SEC-DED code in the conventional uniform manner cannot propagate (i.e., EPR = 0%). Figure 3.8 shows the EPRs under the protection of two independent parity caches whose RARs are 0.58 and 0.30 for checking the I-cache and D-cache, respectively. The results of the four error models are compared. If we assume that error propagation is equally likely to occur in all locations, the expected EPRs with the check codes of these areas would be 42% and 70% (identified by thick dashed-lines in the figures), respectively. However, in most cases, the EPRs are much lower than these values because the distribution of error propagation is not uniform. Organizing the check codes in a cache makes the most of the budgeted area to prevent errors from propagating.

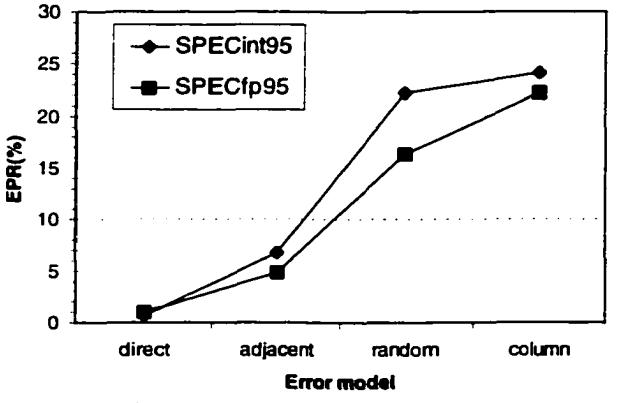
Under the direct error injection model, for all benchmarks except *swim*, the small parity cache allows less than about 3% of injected errors to propagate. This is because this error model and the applications match well with the premise on which parity caching is developed. Due to spatial locality, the parity cache also provides relatively high protection coverage in the adjacent error model. However, we observe larger EPRs on the D-cache for *compress*, *ijpeg* and *swim*. One common attribute of these benchmarks is that their data access does not show good locality, as can be ascertained from their low hit rates in the D-cache given in Table 3.2. The hit rate in the parity cache is even lower. Thus, more items in the D-cache, whose check codes are not present in the parity cache, can be requested. In this case, error propagation takes place unless the items are error-free.



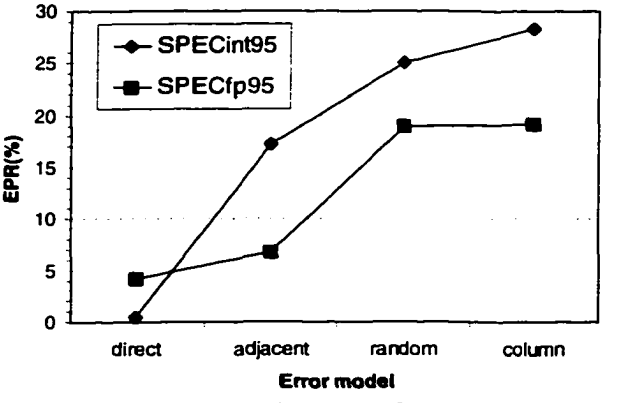
(a) On I-cache



(b) On D-cache



(c) Average EPR on I-cache



(d) Average EPR on D-cache

Figure 3.8 Error propagation rate (EPR).

Even if error occurrence is evenly distributed (i.e., random error model), localized error propagation makes the parity cache area efficient. In the column error model where every line gets an error injection, the results are not very promising. Once a column error injection is executed, any read miss in the parity cache after that results in an error propagation. The column error injection is the worst case test model. Nevertheless, if we consider the area occupancy, on average parity caching provides more protection coverage with a given area as shown in Figures 3.8c and 3.8d.

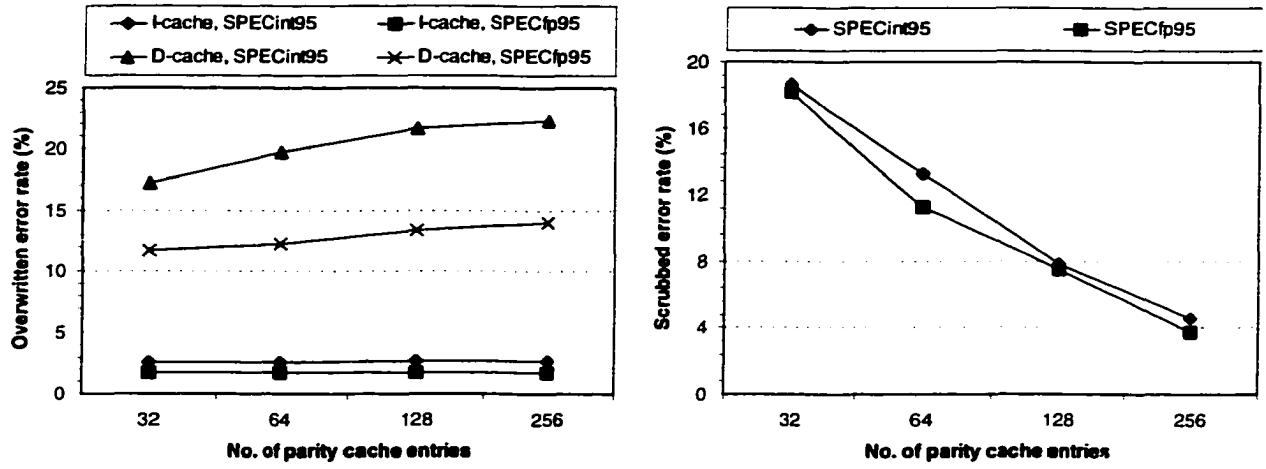
Table 3.3 EPR (%) vs. the number of parity entries.

	Cache type	I-cache				D-cache			
	No. of entries	**32**	**64**	**128**	**256**	**32**	**64**	**128**	**256**
	RAR	**0.13**	**0.20**	**0.33**	**0.58**	**0.07**	**0.10**	**0.17**	**0.30**
Benchmark	compress	0.00	0.00	0.00	0.00	8.33	2.08	0.00	2.08
	gcc	6.84	5.34	2.99	1.28	1.85	1.85	0.00	0.00
	go	7.59	6.29	3.04	1.08	3.91	0.78	0.00	0.78
	ijpeg	6.35	3.94	3.94	1.31	5.30	5.30	6.06	0.76
	li	5.38	1.79	0.22	0.00	0.00	0.00	0.00	0.00
	m88ksim	4.57	4.35	3.48	0.22	0.00	0.00	0.00	0.00
	perl	13.51	8.50	3.27	0.00	3.28	1.09	0.00	0.00
	vortex	12.33	7.71	2.86	1.32	2.36	0.00	0.00	0.00
	SPECint95	**7.07**	**4.74**	**2.47**	**0.65**	**3.13**	**1.39**	**0.76**	**0.45**
	applu	10.75	8.55	3.51	3.07	2.65	1.06	0.00	0.00
	apsi	5.42	3.65	0.65	0.00	14.16	12.79	4.11	0.00
	fpppp	6.64	5.72	4.35	1.83	13.71	5.65	1.61	0.00
	hydro2d	1.36	1.36	1.36	0.91	1.94	1.29	0.00	0.00
	mgrid	0.00	0.00	0.00	0.00	1.52	1.89	0.38	0.38
	su2cor	10.11	9.67	8.35	1.76	0.67	1.34	0.00	0.00
	swim	2.25	0.00	0.00	0.00	29.14	37.09	39.74	41.06
	tomcatv	8.22	8.22	7.31	2.05	2.40	1.80	0.00	0.00
	turb3d	4.55	1.73	1.08	0.00	17.53	5.19	0.65	0.00
	wave5	9.75	4.08	0.00	0.00	3.25	1.30	0.65	0.65
	SPECfp95	**5.90**	**4.30**	**2.66**	**0.96**	**8.69**	**6.94**	**4.71**	**4.21**

Table 3.3 gives the EPRs on the direct error model for an increasing number of check code entries along with RARs. Only 32 entries, which occupy 13% and 7% of the area needed for the uniform structure for the I-cache and D-cache, respectively, bring significantly high coverages. Again, this results from the fact that for many applications the cache access is localized to a very small region. Interestingly, *swim* exhibits a different attribute: as more entries are added, the EPR increases. For *swim*, it turns out that increasing the parity cache associativity is more beneficial than increasing the size. The EPR is reduced to 0.66% on the 16-way set associative parity cache of 256 entries.

Some errors can be removed by normal write operations. Figure 3.9a depicts the portion of overwritten errors out of total injected errors. In the I-cache, only an instruction miss generates a write, while a store request also causes a write in the D-cache. This is why the

overwritten error rate is higher for the D-cache than for the I-cache. The results presented so far are collected from parity caching in combination with error scrubbing (D-cache only). In the case of fewer entries, errors eliminated by scrubbing account for a large portion of total eliminated errors as shown in Figure 3.9b. This is because the number of scrubbing cycles executed per check code entry is larger in a small parity cache. As the parity cache includes more entries, error removal at read access time becomes dominant for most applications.

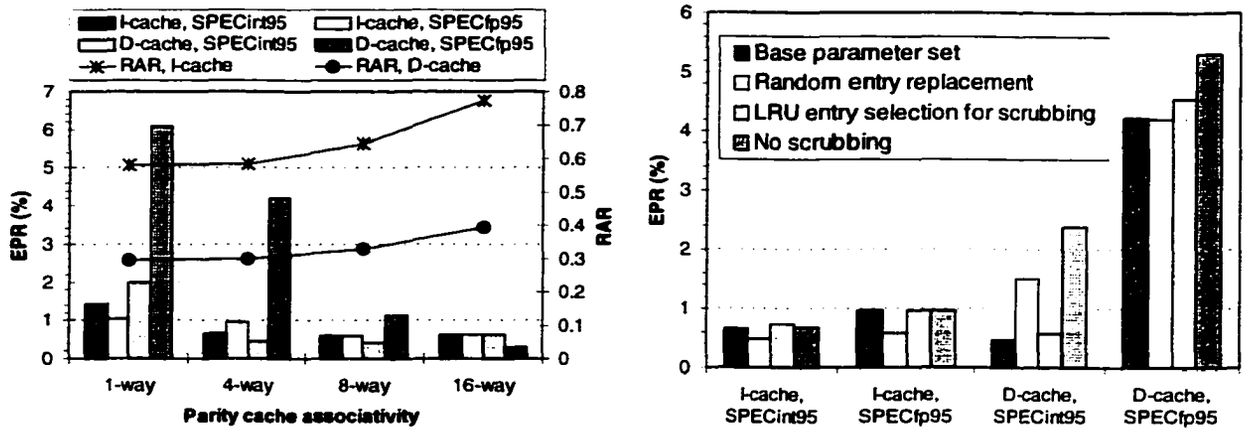


(a) By normal operation (b) By scrubbing

Figure 3.9 Error removal.

We also present the average EPRs for other parameters. From Figure 3.10a we note that higher parity cache associativity enhances the error checking capability. However, the increase becomes insignificant with more than 8-way associativity. On the other hand, area requirement grows rapidly (RARs are plotted on the right vertical axis).

In Figure 3.10b, *base parameter set* consists of LRU policy for entry replacement, error scrubbing, and random entry selection for scrubbing. For performance comparison, three additional simulations were performed with only one parameter variation at a time. Due to the small number of check code entries, one may question if a simpler replacement can affect the coverage. We tested a pseudo-random policy. The LRU strategy performs slightly better on the D-cache for SPECint95. It is, however, a little less efficient than the pseudo-random policy in the other cases. From the results, we conclude that the replacement policy does not significantly affect the performance. In section 3.3.4, we discussed the entry selection issue for error scrubbing. As shown in the graph, the performance gain from LRU entry selection for scrubbing is insignificant. The results without scrubbing are also shown. Scrubbing mostly improves the coverage at the cost of hardware complexity.



(a) Parity cache associativity (b) Base parameters

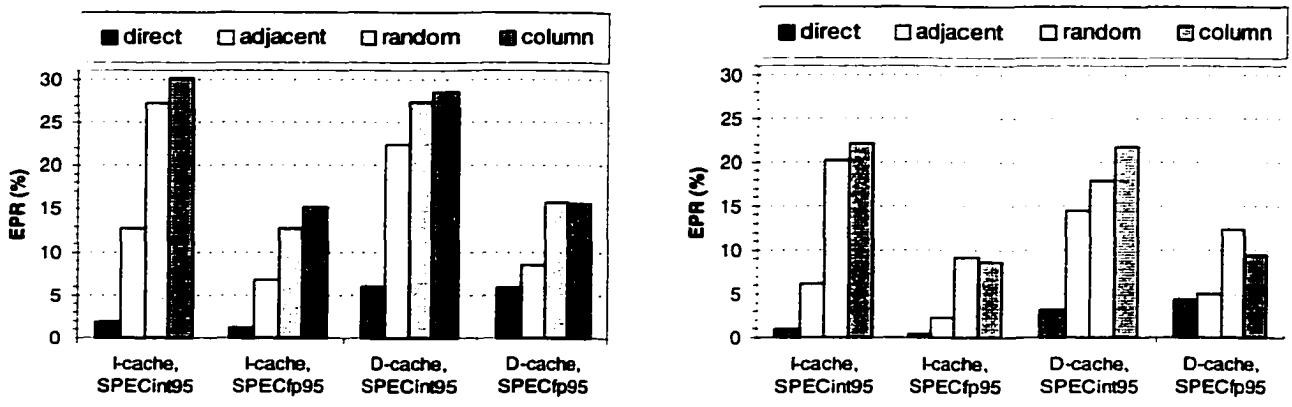
Figure 3.10 Effects of other parameters.

Thus far we have discussed the effect of other parameters on the parity cache performance only in the case of the direct error model. However, similar effects of parameters were noted from the results of simulations under the other three models. We omit them here due to space limitation.

3.5.2 The performance of shadow checking

We have also conducted a set of simulations for shadow checking to investigate how replication architecture with unequal sized modules performs under the presence of soft errors following different error models. Errors were injected in the shadows as well as the main cache. Data items that are supposed to be identical under the normal condition were exposed to independent error injections and are compared for error checking.

Figure 3.11 shows the average EPRs under shadow checking with two shadows. The results for two shadow sizes are compared. Clearly, larger shadow misses fewer errors. Note that the performance variation among different error models are similar to the case of parity caching (Figures 3.8c and 3.8d). The RARs of the shadow cache with 4KB shadows are 0.30 and 0.15 for the I-cache and D-cache, respectively. In the case of 8KB shadows, the RARs are 0.55 and 0.38, respectively. Here, we also observe the EPRs in the direct model are very low, but the same is not the case for the other models. However, we still confirm that shadow checking is very area efficient in all cases. If the designer needs to enhance cache reliability against the types of errors that require replication, but only a small area can be budgeted, then the shadow cache is worth considering.



(a) 4KB shadow

(b) 8KB shadow

Figure 3.11 EPR under shadow checking.

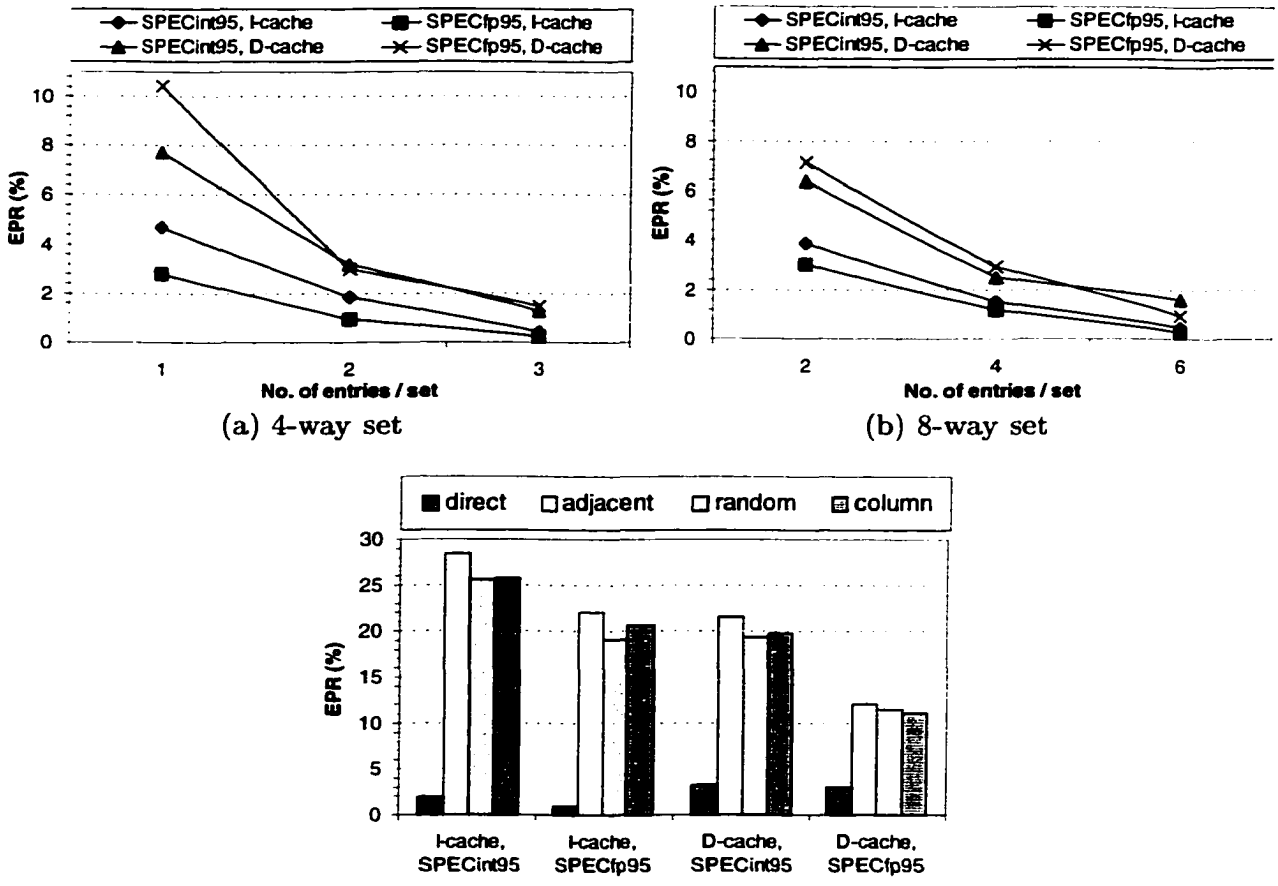
3.5.3 The performance of selective checking

Figures 3.12a and 3.12b show the relationship between average EPRs and the number of entries per set in the direct error model. With only half of the check code required for the uniform structure, EPRs of less than 4% are obtained. However, this is lower coverage than a parity cache of the same area can provide. The reason for this is that in selective checking, a line is selected to assign check codes within the scope of a set rather than the entire cache. Recall that an advantage of selective checking is that it only needs a simple modification to the conventional architecture.

From Figure 3.12c, unlike the first two checking schemes, we note that the protection coverage of two check codes per 4-way set varies very little under the three error models. This is also due to the fact that the check codes are managed independently for each set. Although EPRs are relatively high in the three models, they are still lower than 29%, which is much higher coverage than an intuitive expectation with about a half size check code array. This indicates that our locality-based checking scheme efficiently uses the given check code area.

3.6 Summary

In conventional architectures, as the size of the primary cache grows, the redundant code for data integrity checking also needs to be increased proportionally. We have proposed new architectural solutions for the situations where enough area cannot be budgeted to support this uniform organization and further expansion of the protection code. In our schemes, check code



(a) 4-way set

(b) 8-way set

(c) Effect of four error models

Figure 3.12 EPR under selective checking.

can be designed for a given area in such a way that the most frequently accessed cache lines, which are likely to be the most error-prone and are likely to have the lowest error propagation latencies, take precedence in integrity checking over less frequently used lines. Prediction for line selection is performed by taking advantage of locality in cache accesses.

We have considered four possible error models and applied them to our simulated systems. From the simulation, we have found that with α% check codes of the uniform error checking architecture, the proposed schemes achieve far more than α% in error protection coverage. In particular, significantly low EPRs are obtained under the direct error model with a small area. We have also shown that adding the error scrubbing technique is more beneficial for a system with a small number of protection code entries. Parity caching and shadow checking schemes are more effective in the adjacent error model than selective checking. However, selective checking requires the simplest organization and management, and is thus easy to implement for multi-way set associative caches. Despite the unbiased error injection in time and location

in the random and column models, our schemes that are tuned for the protection of the MFU lines are still area effective for such error models.

We have shown that our locality-based configuration schemes for the check codes can be adapted to current systems with a small overhead. An important advantage of the proposed architectures over the conventional uniform structure is the flexibility given to the system designer in planning cache systems of the desired capacity in terms of size and reliability. We have given an area estimate for the schemes based on an area model. In order to fully validate the benefit of the schemes obtained from controllability of area occupancy, the geometry of physical chip area in the VLSI design needs to be investigated further.

CHAPTER 4 Cache Write Verification in Multi-Level Caching Systems

Due to the nature of reference locality, the CPU mostly communicates instructions and data with the first level on-chip caches. These are originally fetched from the secondary cache or memory with very low frequency. Thus, the guarantee of this initial *fetch-and-write* into the first level cache, which is rare but fundamental for correct future operation, is indispensable for a dependable caching system. This chapter presents a new cache write error detection scheme, called cache write sure or CWS, which exploits the preexisting information redundancy of the multi-level caching systems. The effectiveness of this detection technique is evaluated by using an on-the-fly trace driven simulation for thirteen benchmarks combined with software error injection. The results show that for most workloads, the CWS provides almost complete write error detection for a non-protected I-cache in a two-level on-chip caching system with a cache cycle time ratio between L1 and L2 of 1 to 5. At the same time, it can also significantly enhance write error protection for the D-cache using the same hardware.

4.1 Importance of Initial Data Transfer

The frequently used portion of instructions and data is maintained in fast cache memories so that the processor can communicate with the caches rather than with slow main memory for most of the program execution time. As a larger portion of silicon area is devoted to on-chip caches, a higher probability of fault occurrence in the cache memory can be expected. Instructions or data items in the on-chip cache are likely to be accessed repeatedly until replaced in the case of a miss. If a certain transient fault corrupts the contents of a cache line during its initial write operation (fetching from the lower level memory or cache and copying to the on-chip cache), the subsequent references to the line would be erroneous and can cause further error propagation. In the case of a cache miss, memory components in both levels must operate properly for a correct initial write operation. By contrast, the CPU accesses only the on-chip cache on a cache hit. In practice, since cache hit ratio is considerably high for most applications with common on-chip cache sizes (e.g., 8-KB~2-MB), the frequency of this initial

data write is very low. Therefore, adequate protection against errors during the initial write operation is the first essential step in enhancing the dependability of the cache system.

One common approach is to employ ECCs. In general, ECC is constructed across a certain number of bits in order to correct single-bit errors or detect double-bit errors. It is also possible to build an ECC for multiple-bit correction and detection, but the corresponding complexity of the circuitry and the amount of redundancy required increases excessively. Hence, ECC is not practical if a large number of bits or burst line errors are expected.

A bus protocol, called echo-back [90], was proposed to detect faults generated during cache operation. If either the bus master or bus slave writes an address on the bus, the other one reads it and then writes the sensed address back on the bus to verify the address transfer. The validation of the data transfer between the master and slave is accomplished in a similar manner. The protocol is particularly useful in avoiding latent faults and multiple faults. However, the single cache read/write cycle time needs to be increased for the echo-back. A minimum degradation in the system performance under this protocol can be obtained only when the cache accesses are consecutive so that the verification time can overlap with the normal bus cycle.

In [91], we developed an error detection mechanism, called shadow caching, to overcome the limitations of conventional ECC. A 0.5~2 KB cache keeps the most frequently used portion of main cache lines and detects transient errors that have been generated during any operational phase including the initial write stage, which is of our specific interest here.

In this chapter, we present a new hardware cache write protection scheme, named as *cache write sure* or *CWS*, in the multi-level cache systems using a FIFO type verification queue. To ensure that the initial write operation in the first level cache has been performed correctly, each newly updated line is also copied into a verification queue at the time of its first read access. While the CPU proceeds with its operation using the information from the first level cache, denoted by *L1 cache*, the cache controller performs the line verification cycle by comparing the corresponding data in the queue and the secondary cache, denoted by *L2 cache*. The CWS scheme is developed to remedy the insufficiency of ECC in cache write error protection. The scheme requires far less chip area and complexity than ECC.

4.2 Motivation

As VLSI chips become denser, a larger chip area can be made available for the cache system. However, the monotonous increase in cache size or associativity cannot be an absolute solution because larger caches become slower. Multi-level cache hierarchy, therefore, is widely used [92], [93]. Some microprocessors even employ the two-level caching approach on a single chip [94]. For example, 96 KB and 32 KB L2 caches are implemented on the DEC Alpha 21164 and PowerPC x704 processor chips, respectively.

In the memory hierarchy, the main memory includes all the data of the upper level memories. Similarly, L2 cache typically includes all the information of L1. Notice that the information redundancy is employed for system performance improvement. The basic idea of our work, the CWS scheme, is to exploit this existing redundancy for cache fault tolerance purposes. We compare the contents of a verification queue copied from L1 cache against the contents of L2 cache to check if the data items of an L1 line, which will be used many times by the processor, have been correctly fetched and written into the top level cache at the outset. This faithful incipient fetch-and-write must first be guaranteed for reliable future references.

Table 4.1 Hit ratios (%) of 16 KB on-chip I-cache and D-cache.

Benchmark name	I-cache	D-cache
am2	99.996	97.816
intmm	99.999	98.590
num.conv	99.999	99.997
pla	99.999	98.144
roll.com	99.996	99.993
state	99.998	99.998
dhrystone	99.999	98.602
fft	99.999	49.163
flops	99.998	99.990
hanoi	99.997	99.995
linpack	99.999	91.883
mm	99.999	89.292
nsieve	99.999	86.846

Table 4.1 shows the hit ratios of the on-chip instruction cache (I-cache) and data cache (D-cache) for thirteen benchmarks on a DEC Alpha AXP 3000. Each L1 cache provides the majority of requested program instructions and data in all cases except for *fft* execution. Specifically, in the I-cache, there is an extremely small number of initial fetch-and-write operations since the hit ratio averages more than 99.9%. This indicates that once an I-cache line, which includes multiple instructions, is fetched, it is read many times due to the nature of instruction

reference. Note that most of the working set of instructions used during program runtime is not fetched from the L2 cache or memory, but from the L1 cache. Thus, during this period, the L2 cache can be accessed by other system components independent of the processor's operation. By taking advantage of this idle period, the CWS technique will perform information comparison between the two copies of a cache line in the verification queue and in the L2 cache, thereby verifying the validity of the initial fetch-and-write operation.

In the case of the D-cache, data write occurs more often than with the I-cache because the processor may repeatedly update the data of a D-cache location after the initial fetch. Moreover, it also has a slightly higher miss ratio than the I-cache in many cases. Therefore, we can expect that the number of verifications required to detect data write errors is higher. One interesting question here is whether or not the L2 cache is actually available for every data verification. In addition to the L1 misses, the processor's intrinsic write operations to the D-cache also increase the L2 cache traffic if a write through L1 cache is employed. We intend to answer this question with our simulation study. The CWS scheme targets almost complete protection for the I-cache. Using the same hardware resource, it also significantly enhances data integrity for the D-cache.

4.3 Baseline Architecture

Figure 4.1 shows the baseline architecture used to implement the CWS scheme. It consists of a verification queue, called *write sure queue* (WSQ) and a verification unit with look through L1 caches. The processing unit corresponds to a pipelined RISC CPU core. To allow parallel instruction fetches and data accesses, the L1 cache is conventionally separated into the I-cache and the D-cache. Those caches themselves are not ECC- or parity-protected. Since the I-cache does not copy back any instruction to the next level memory, no write strategy is required. In the proposed CWS scheme, the data coherency between the L1 and L2 should be maintained all the time because the data in the L1 is verified against the data in the L2. Thus, the D-cache uses buffered write through and write around policies that are easier to implement than write back with write allocate or invalidate [95], [96], which cannot guarantee the data coherency in the memory hierarchy.

If a write hit occurs in the D-cache, both L1 and L2 caches are updated. The mixed L2 cache with write back and write allocate policy can be either on- or off-chip memory and is a superset of the L1 cache without ECC protection. In order to minimize CPU wait time due

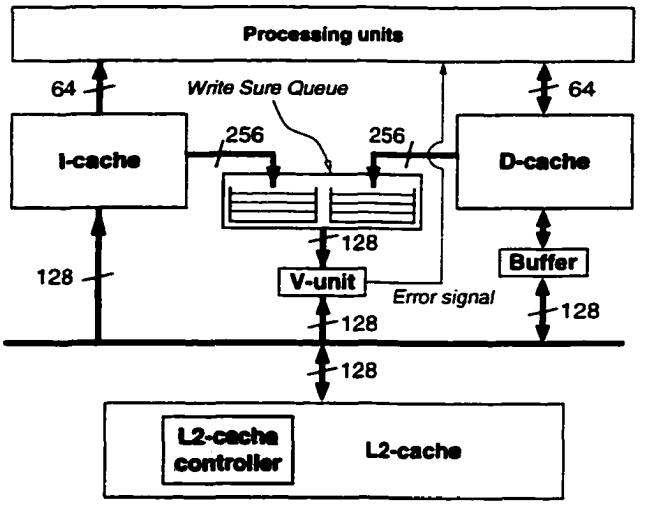


Figure 4.1 Baseline architecture for the cache write sure scheme.

to the write operation to the L2, write queues are employed. This allows the CPU to continue its operation while the cache controller performs the data update. In case of a write miss on the D-cache, the incoming data item is written to the L2 cache directly with no change to the D-cache. As a result, an L1 cache write operation occurs only on a read miss or a write hit.

Only two extra components, the WSQ and the verification unit, are added to a conventional non-protected cache system on the processor chip. We assume that the L2 data is error free because the goal of the CWS system is not the detection of L2 data errors but the data integrity checking in the L1 caches generated by transient faults in buses, control lines and/or RAMs during fetch-and-write operations. The L2 data itself is always considered correct, but it is possible for the L2 data items to become erroneous during transfer to the L1. The WSQ is a FIFO device, and its dual input/single output external configuration makes it possible for both L1 caches to copy their tag and data to the WSQ at the same time. The queue internally consists of two buffers and provides only a single item per verification cycle to the verification unit. To increase integrity of tag and data items after the copy from the L1 cache, parity bit protection is used for each queue entry. An error signal line is provided by the verification unit to inform the CPU of comparison result.

4.4 Instruction and Data Verification

In our baseline model, the CPU requests an instruction fetch from the I-cache every cycle. When a new instruction or data item is written to the L1 caches on a read miss or write hit,

an additional cache line status bit, denoted by *verified bit*, is set to zero in order to indicate that a cache line has been updated and its verification is required. During this operation, the CWS system performs no operation.

Once the line is in the L1 cache, it is likely to be used in the near future because of reference locality. When an item (or subblock) of the line is requested, it is sent to the CPU without any delay. In the mean time, by sensing that the verified bit is zero, the tag and data portions of the entire line, which may or may not have been corrupted, are copied into the WSQ and the verified bit of the line is set to one. In addition, parity bits are computed and attached to the queue entry on the basis of a subblock. It is also possible to implement more secure protection code for each WSQ entry. In any case, the computations of protection bits are always accomplished in parallel, and therefore, no impact on critical path is incurred. The parity codes are used to check the integrity of the line copy in verification phase.

Due to the limited capacity of the WSQ, the line copy may not always be possible. We denote the condition in which the queue entries are all occupied as *queue full state*. One possible solution is to make the processor wait until the verification unit finishes one line's verification, which will make room for one more entry at the tail of the queue. However, this increases overall program execution time. Since the minimization of the system performance impact is one of the goals of the proposed CWS scheme, the line copy-and-verification is simply postponed until a sufficient condition of the line copy, i.e., the tail of the queue is empty on its reference, is satisfied. In this case, the verified bit and queue status are not changed. The drawback of this strategy is that a line's verification can be completely missed if no other occasion arises for the verification before its replacement. To reduce the probability of the queue full state, more queue entries can be added, but it may or may not be really effective depending on the access pattern. We will discuss the effect of queue size on the error detection coverage of the CWS scheme based on the simulation results in Section 4.6.3.

A new L1 cache line is verified only one time. Once the new line is copied into the verification queue, its recopy is prevented by monitoring the verified bit on subsequent accesses. Since the L1 cache has the look through configuration, the bus between L1 and L2 caches is busy only in the event of an L1 miss and a write hit on the D-cache. A missing item in L1 cache is fetched from L2 or main memory and provided to the CPU and the appropriate L1 cache at the same time. If a requested instruction or data is found in one of the L1 caches, the L2 cache controller independently performs the data comparison between queue head and corresponding L2 line. A single line verification is achieved in two half entry verification cycles. If any error is

detected in the tag portion of the queue head when the parity code is checked, the verification is aborted and the entire entry is removed. If the stored parity bit does not match with the calculated parity of a subblock, the result of the subblock comparison is ignored during the verification cycle.

The data verification of each half of the queue head takes longer than an L1 cycle time due to the relatively slow speed of the L2 cache. The cycle time for the L2 cache and a verification cycle are longer than the L1 cache cycle time. Since the L2 is larger in size, we assume that multiple L1 cycles are required for a single L2 cycle. The L2 cycle time consists of the following two components:

$$\begin{aligned} t_{L2\ cycle} &= t_{L1\ miss\ penalty} + t_{L2\ access\ time} \\ &= \tau_{L1} + k\tau_{L1} = (1+k)\tau_{L1}. \end{aligned}$$

where τ_{L1} is L1 access time, $t_{L2\ access\ time}$ is the sum of $t_{L2\ probe} + t_{data\ transfer}$, and k is an integer larger than 1. In general, k is 2~5 and 5~10 for on- and off-chip L2 caches, respectively. In DRAM main memory case, k is normally 10~50. Since the verification time does not include $t_{L1\ miss\ penalty}$ and is bounded by the L2 cache speed, a period shorter than $t_{L2\ access\ time}$ is assumed for the verification cycle for each half of the full line verification. This half verification time is denoted by τ_{HV}.

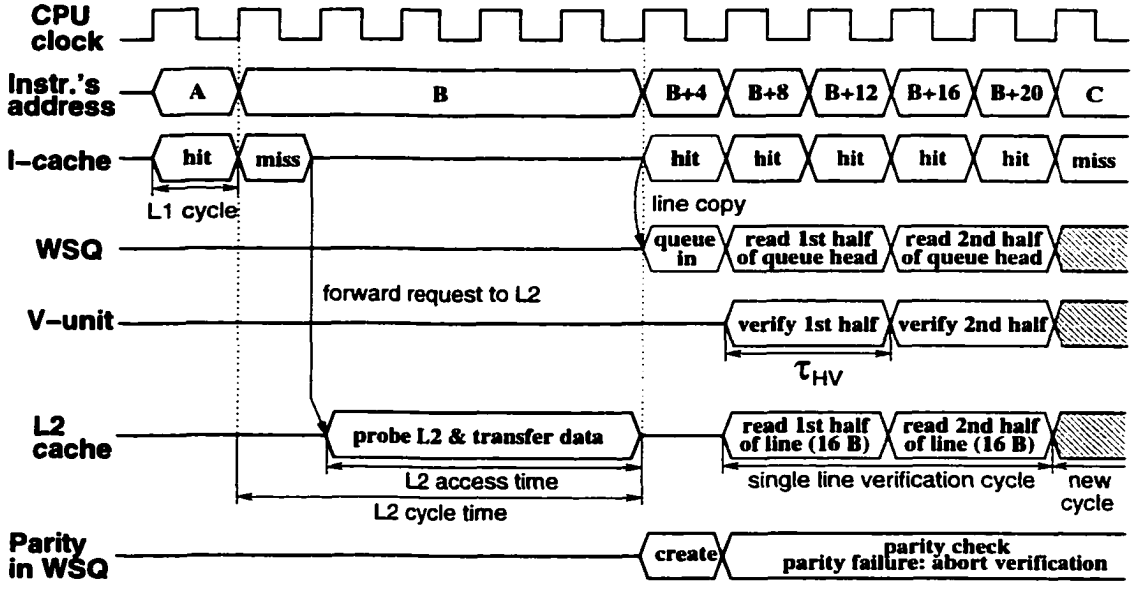


Figure 4.2 A timing diagram example of instruction verification.

Figure 4.2 illustrates a timing diagram example of the CWS operation for the I-cache where k and τ_{HV} are equal to 4 and $2\tau_{L1}$ each. A read miss on the I-cache generates a cache line fetch from address B of the L2 cache and a write into an L1 line according to the address mapping. On the next read cycle, the requested instruction's address is B+4, and the whole line (B~B+31 = 32 Bytes) is copied to the WSQ and is used for the instruction verification during the following four clock cycles in this example as the L1 provides instructions to the CPU without accessing the lower level memory. An entire line verification consisting of two half verification cycles ($4\tau_{L1}$) is accomplished successively in this case. However, each of the half verification cycles (τ_{HV}) need not be adjacent. If the system bus and L2 cache are not free after the first half verification, the second half verification is postponed until the next available bus cycle. The parity bits, generated during line copy phase, are checked in each half verification cycle.

An L2 cache access request may arrive from the processor while a verification cycle is in progress. In this case, two approaches, called preemptive and non-preemptive, are possible. Under the preemptive scheme, the CPU request overrides any ongoing verification cycle. The interrupted verification is resumed at the next available L2 idle cycle. In the second case, the ongoing verification cycle always continues. This increases the program execution time due to the increase in CPU wait cycles. However, it might be worth the delay if this expense prevents the CPU from using incorrect data or instructions. Besides, the preemptive policy is generally more complicated to implement than the non-preemptive one because it requires extra mechanisms for rescheduling. Once a verification cycle is interrupted, the data and tag in the queue have to be maintained at the same value. In this chapter, we examine and compare the detection coverage and overhead of both schemes.

4.5 Evaluation Methodology

We evaluated the proposed CWS scheme based on an on-the-fly trace driven simulation method using the ATOM tool [97]. The instruction trace of the entire program is instrumented for thirteen benchmarks on the Digital Alpha AXP 3000 Model 400. Table 4.2 shows the program description and instruction type distribution of each benchmark used.

A direct-mapped cache was used for each L1 cache as recommended in [98] and 32 byte-lines were assumed for both the L1 and L2 caches. In order to obtain the advantages of two-level caching such as reduction in cache access time and dynamic allocation of instructions and data

Table 4.2 Characteristics of benchmark programs.

Benchmark name	No. of instructions	Load (%)	Store (%)	Workload Description
am2	10.5M	37.0	6.6	iterative control loop programs from Boeing airplane company
intmm	97.9M	17.4	5.9	
num.conv	36.0M	21.6	9.6	
pla	71.0M	50.0	7.2	
roll.com	9.4M	32.3	14.2	
state	34.1M	36.3	14.7	
dhrystone	58.8M	23.6	12.9	synthetic program intended to be representative for system integer programming
fft	104.6M	12.7	11.9	fast fourier transform
flops	108.3M	2.5	3.2	estimating MFLOPS rating for specific FADD, FSUB, FMUL, and MDIV instruction mixes (double precision)
hanoi	20.2M	25.0	20.4	integer program to solve the Towers of Hanoi puzzle using recursive function calls
linpack	565.0M	28.7	14.8	linear algebra routines of 100 × 100 matrices (rolled double precision)
mm	683.4M	19.8	4.7	matrix multiplication (500 × 500 standard size) from T. Maeno
nsieve	878.8M	12.7	9.6	integer program to generate prime numbers using a method known as Sieve of Eratosthenes

(i.e., the ratio of instructions to data within a single L2 cache can be flexibly determined), the L2 cache size has to be much larger than the L1 cache. Since the verification cycle depends on the cycle time of the L2 cache rather than its size, we accommodated the size flexibly in accordance with the cycle time. We used two types of L2 caches and a main memory model in the memory hierarchy to examine the effectiveness of the CWS scheme. The L2 types and their cycle times used are two on-chip caches with cycle times of $4\tau_{L1}$[1] and $7\tau_{L1}$, two off-chip or external caches with cycle time of $13\tau_{L1}$ and $15\tau_{L1}$, and an external main memory of $61\tau_{L1}$ with no L2 cache.

To estimate the detection coverage of the CWS, we injected errors while a program was running. The error injection rate (EIR) was empirically determined to be about 0.1% and 0.001% out of total number of write operations for I-cache and D-cache, respectively. Table 4.3 shows the write rate on each L1 cache computed by

$$write\ rate\ on\ I\text{-}cache(\%) = \frac{number\ of\ read\ miss}{number\ of\ read\ miss\ +\ hit} \times 100,$$

$$write\ rate\ on\ D\text{-}cache(\%) = \frac{number\ of\ read\ miss\ +\ write\ hit}{number\ of\ read\ +\ write} \times 100.$$

Since the write frequency on the I-cache is very low, the actual number of errors injected is also very small. This reflects rare error occurrence in real life. To measure the performance of

[1] τ_{L1}: L1 cache cycle time

Table 4.3 Write rate on L1 cache.

Benchmark name	I-cache (%)	D-cache (%)
am2	0.0003	13.15
intmm	0.0004	25.49
num.conv	0.0010	30.77
pla	0.0005	14.41
roll.com	0.0043	30.60
state	0.0011	28.82
dhrystone	0.0008	34.43
fft	0.0006	82.91
flops	0.0015	55.97
hanoi	0.0027	45.00
linpack	0.0004	40.88
mm	0.0001	28.91
nsieve	0.0003	56.19

the CWS, we mainly used write error detection coverage (WEDC), which is defined as

$$WEDC(\%) = \frac{number\ of\ errors\ detected}{number\ of\ errors\ injected} \times 100.$$

As discussed in Section 4.4, both bus management schemes, preemptive and non-preemptive, have trade-offs between the detection coverage and the CPU stall/scheduling overhead. Hence, for comparison purposes, we obtained the verification aborting ratio (VAR) for the preemptive handling scheduling and the non-preemptive handling overhead, CPU stall per instruction (SPI). Each metric is computed as follows:

$$VAR(\%) = \frac{number\ of\ verification\ aborted}{number\ of\ L1\ cache\ misses + D\text{-}cache\ write\ hits} \times 100,$$

$$SPI(\%) = \frac{number\ of\ CPU\ cycles\ stalled}{number\ of\ instructions\ executed} \times 100.$$

4.6 Simulation Results

To show the effects of each parameter on the detection capability, we varied only one parameter at a time. Basic parameter values are 16 KB L1 cache, preemptive bus handing, $\tau_{HV} = 2\tau_{L1}$, pipelined processor with seven stages, and four entries in the WSQ.

4.6.1 Effect of secondary memory type

Figure 4.3 illustrates the WEDC for I-cache with three types of secondary memories. The CWS scheme provides 100% coverage except for the two applications in the case of an on-chip L2 cache with $\tau_{HV} = 2\tau_{L1}$. Note that the number of errors injected is very small, and thus the failure of a single error detection results in a very large decrease in the detection percentage.

To obtain a more accurate detection rate, we can compute the average value of many runs with different error injection times or line locations all at the same value of EIR. However, this approach is time-consuming. As an alternative method, a heavy EIR can be used to compare the WEDCs although the error occurrence pattern is far from our expectation of the real situation. According to the simulation results, 26.6% increase and 1.1~10.8% decrease in WEDC was observed on *num.conv* and other workloads under around 10% increase in EIR, respectively. This difference in detection capability comes from the fact that the sensitivity of WEDC with respect to EIR becomes larger as the write rate on the L1 cache becomes smaller.

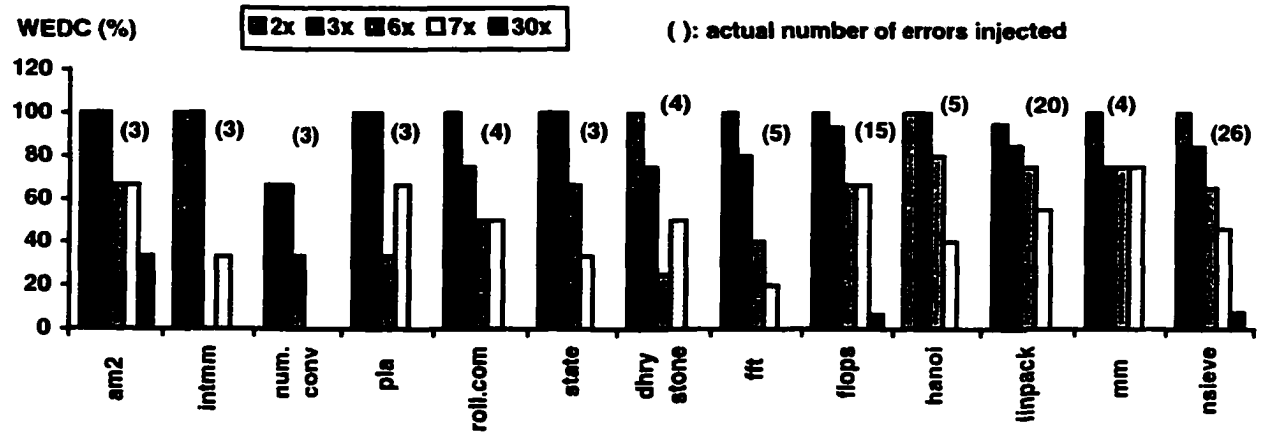


Figure 4.3 Effect of half queue verification cycle time on I-cache.

When off-chip L2 is used ($\tau_{HV} = 6\tau_{L1}$ or $7\tau_{L1}$), the WEDC fluctuates between 0~80%. The chip-to-chip communication is much slower than on-chip transmission. Obviously, the longer verification cycle can make the WSQ full, i.e., arrival rate is greater than the service time for the queue. In the case of external memory ($\tau_{HV} = 30\tau_{L1}$) as the instruction comparison source, the latency is very high in comparison to the L1 cycle time. Thus, no error detection is obtained in most program executions.

Both L1 caches were always included in the memory system at each simulation run. As we observed in Table 4.3, data writes in the D-cache accounts for a substantial portion of the cache operations. Since a single WSQ is shared by the both L1 caches, we expected some impact of the D-cache write traffic on the WEDC for the I-cache. However, the CWS shows 100% WEDC for the I-cache on eleven benchmarks in an on-chip L2 configuration, although the D-cache provides more cache lines to be verified due to its frequent write operations and its increased bus traffic between L1 and L2. From this observation we can draw an important feature of the CWS: the WEDC is independently determined by each cache's access pattern.

The D-cache might update its cache lines before they are verified in the event of additional write hits. This is because the real verification cycle always has a certain amount of delay after a read request arrives. The L2 cache controller needs to wait to start a new verification cycle regardless of the bus handling scheme. During this interval, new data might overwrite the error in the L1 cache. In this case, the corresponding entry of the WSQ must be also updated to prevent a misdetection. Furthermore, we could observe that many errors injected on D -cache are quickly removed because of both frequent write hits and read misses, where the corrupted line is overwritten by new data before the line is verified.

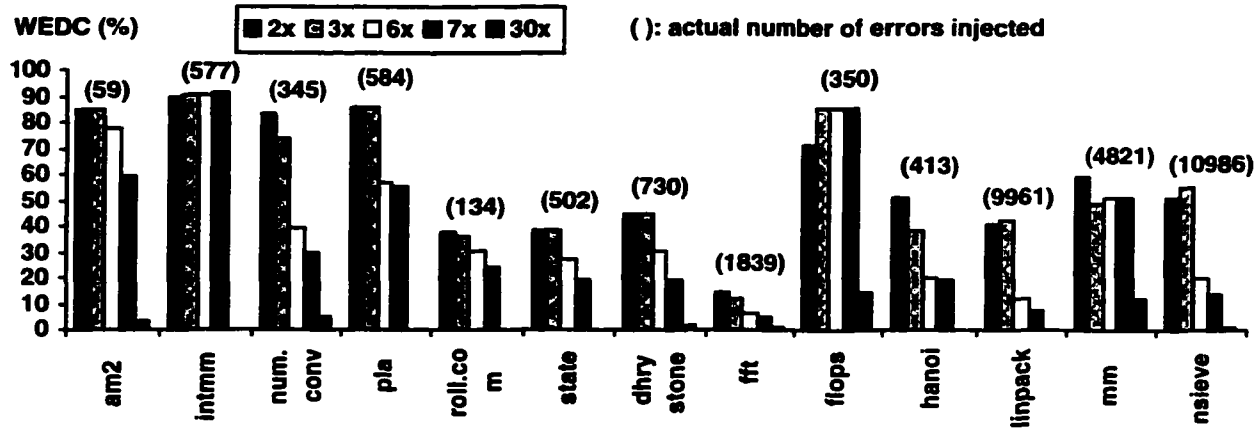


Figure 4.4 Effect of τ_{HV} in accordance with three L2 types associated with D-cache.

Figure 4.4 depicts the WEDC for the D-cache for various secondary memory systems. Even on-chip L2 cache with $\tau_{HV} = 2\tau_{L1}$ shows poor error detection performance in many applications. Nevertheless, the CWS presents on an average of 57.9% WEDC on the D-cache using common system resources such as the WSQ and verification unit that are also used for the complete error detection on the I-cache.

4.6.2 Effect of L1 cache size

In general, as the L1 size becomes larger, its hit ratio also increases. Although increase in hit ratio obtained may be insignificant, the improvement in cache access time can be notable. The higher hit ratio allows the WSQ to achieve higher access to the L2 bus for verification. For several applications, Figure 4.5 shows this expected behavior. One reason of exceptional results on the I-cache for *num.conv*, *hanoi*, and *linpack* is that the number of errors injected is very small as discussed in the previous section.

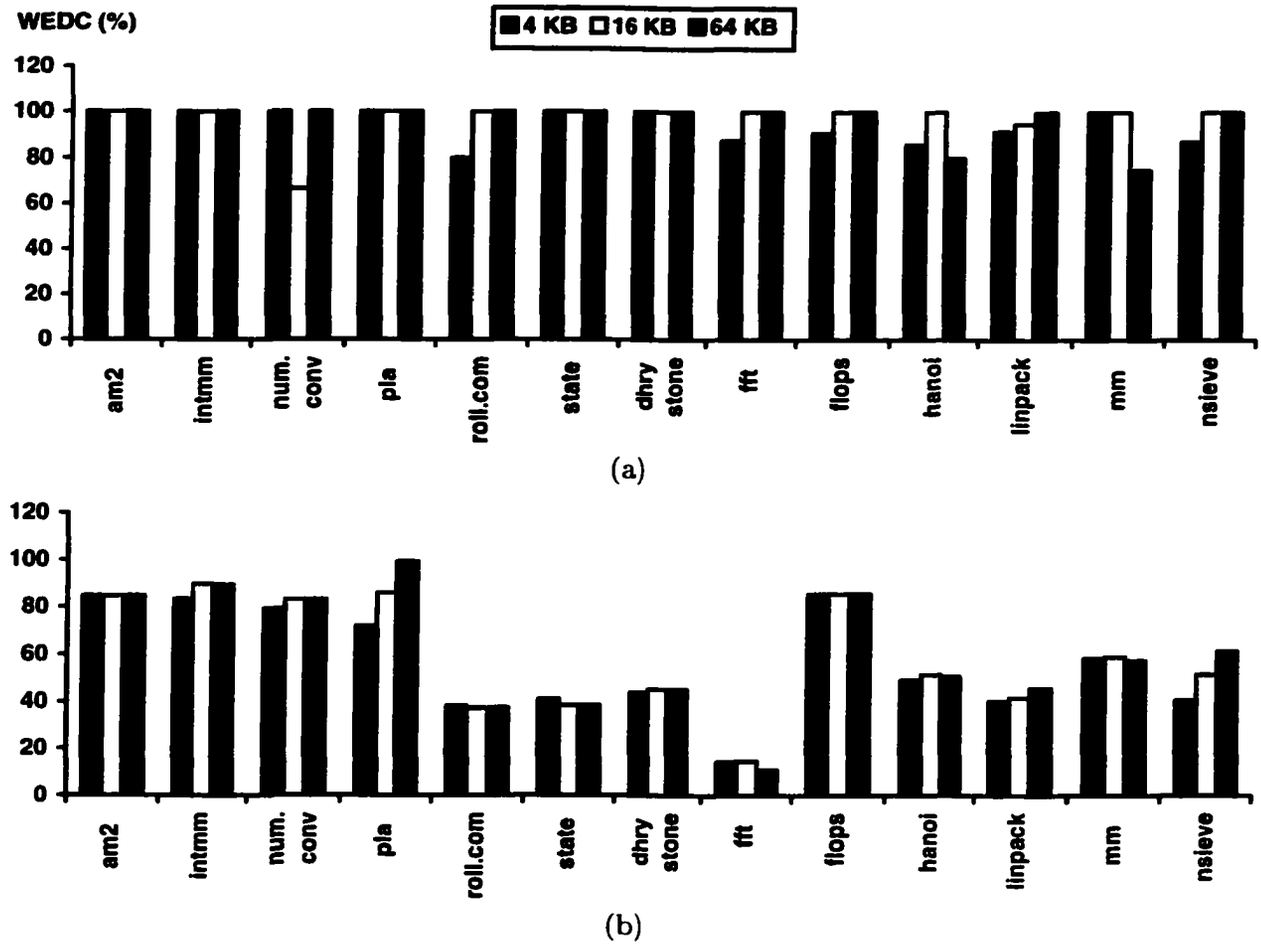


Figure 4.5 Effect of the cache size: (a) I-cache. (b) D-cache.

In the case of the D-cache, the WEDC of larger caches is not always higher than that of smaller ones because different size caches cause distinct streams and frequency of data writes. The perfect error detection is possible only if all the verification requests from the L1 caches are performed successfully. If most of the requests arrive in a small interval, L2 is not available for the verification. The newly updated cache line may not be verified. Smaller caches generate more read misses and frequent data replacements while the larger caches provide a higher write hit ratio. More read misses or write hits on the cache require more frequent data verification. Thus, the congestion in verification unit and L2 bus may happen in any size of D-cache.

In the two-level on-chip caching system, the L1 cache has a limited size because more chip area is needed for the L2 cache. A 16 KB or 32 KB is a common size for an on-chip L1 cache. Note that the CWS presents complete WEDC for 16 and 64 KB I-caches with L2 cache cycle times of $5\tau_{L1}$ and $\tau_{HV} = 2\tau_{L1}$ for most applications under 0.1% of EIR.

4.6.3 Effect of WSQ size

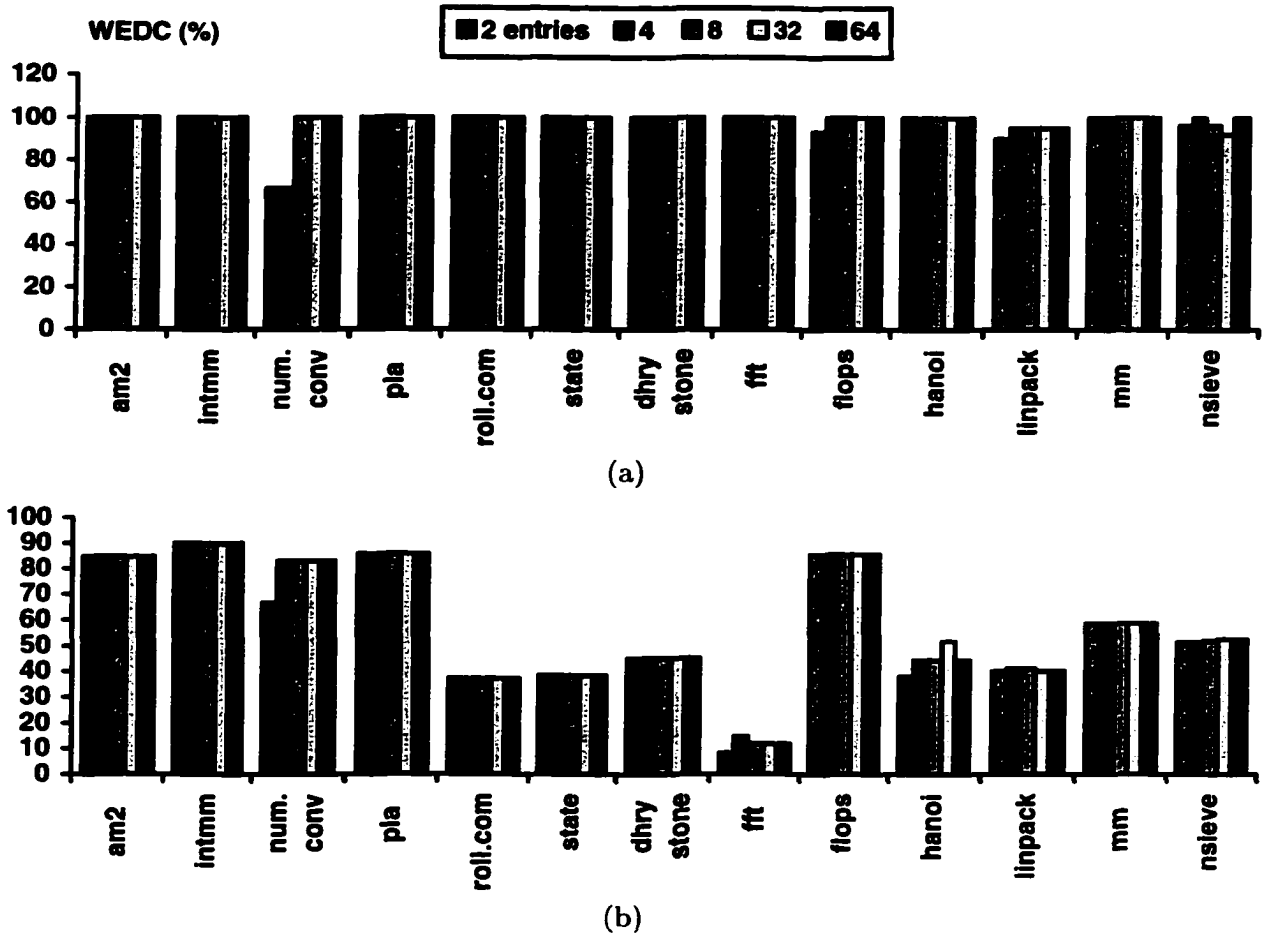


Figure 4.6 Effect of WSQ size: (a) I-cache. (b) D-cache (each column of a benchmark indicates a different number of entries in the WSQ).

One may think that the number of valid queue entries is a major factor in achieving higher WEDC. If the queue entries are not promptly verified and cleared, or if the queue size is too small, new cache lines cannot enter the queue for verification, which corresponds to the queue full state. Even if the queue has a large number of entries, it is still possible to reach the queue full state if the service time is slow. As a result, the queue size is not a critical parameter. To achieve a higher WEDC with a small number of queue entries, verification requests from the L1 caches must be evenly distributed during the program execution and minimum number of L2 bus cycle must be requested by the processor or L1 cache misses.

We conducted a set of simulations with different queue sizes to examine its effect on error detection as shown Figure 6. For most applications, there is no remarkable performance

difference among various queue capacities. The results show that the verification performance of the CWS mostly depends on available L2 bus cycles rather than the queue size. The advantage of a larger queue is achieved only when a new cache line is copied and verified, as a smaller queue cannot accept the request due to the queue full state before the line's replacement. Many entries in a queue might increase mean time to detection (MTTD) in a certain case. The verification of a new line in the queue tail takes a longer time right after a queue full state in a large queue because more lines in the queue need to be verified first. From our simulation results, in the off chip L2 configuration ($\tau_{HV} = 6\tau_{L1}$), a 2-entry queue provides slightly better (on average 4.37%) error detection than a 64-entry queue. However, for most cases, a WSQ with only four entries accomplishes almost same error detection coverage as the larger capacity queues and occupies less chip area.

4.6.4 Effect of bus management and degree of pipeline

Table 4.4 Effect of bus management scheme.

Cache type	I-cache		D-cache	
Benchmark	num.conv	nsieve	mm	fft
preemptive	66.70%	100.00%	59.10%	14.70%
non-preemptive	100.00%	96.20%	51.80%	9.30%

While the L2 controller performs the verification cycle with the data from the WSQ, no L1 cache misses are desirable if the bus is to remain free. If any L2 data or instruction request from the processor is issued, the controller has to decide to either continue the ongoing verification or to abort it, depending on system design requirements. We measured the WEDCs of two approaches, preemptive and non-preemptive. Different detection coverages were observed in only few cases as shown in Table 4.4 On the *num.conv* benchmark, non-preemptive detects all three errors injected during 36 million instruction executions with the SPI of 0.023 while the preemptive scheme provides little higher WEDCs on three applications. One possible reason for this is that in a non-preemptive mode, the holding time for an item is shorter than in the preemptive mode. The different queue holding time can change the sequence of verifications for newly written cache lines and thus result in slightly different detection coverage.

Table 4.5 shows the VAR on each benchmark. It varies in a wide range along applications. Higher VAR indicates more overhead for resuming aborted verifications. The selection between these two schemes is not straightforward due to the different overhead nature of the two which

is beyond the scope of this chapter. But the system architect can make a decision based on preference because both handling methods show similar detection coverage in most cases.

Table 4.5 Verification abortion rate (%) on preemptive bus handling.

Benchmark	VAR (%)
am2	8.708
intmm	0.633
num.conv	6.955
pla	11.264
roll.com	3.001
state	13.840
dhrystone	7.897
fft	6.897
flops	0.032
hanoi	22.220
linpack	24.583
mm	15.937
nsieve	6.079

To study the effect of the degree of pipeline for the processor, five and nine stage pipelines were examined in addition to one with a degree of seven. The change in degree of pipeline may change the bus request timing because the D-cache access is shifted to a different CPU clock cycle after the instruction is fetched. However, it turned out that there is no difference in the WEDC on either cache in all three cases.

4.7 Summary

We have presented a new cache write error detection technique in multi-level caching systems. The CWS scheme detects all of the errors injected in the on-chip I-caches under the configuration of the on-chip L2 cache whose access time is no longer than double the L1 cycle time. Although the detection of corrupted data in the D-cache is limited, write error protection is simultaneously applied to both L1 caches due to their insignificant conflict in verification requests. Many microprocessors are equipped with an integral of multi-level caches because it is not uncommon for the system performance to suffer significantly without an effective cache hierarchy. Under this trend, the CWS can be more effective in write-and-fetch error detection for non- or insufficiently protected on-chip L1 caches.

Simulation results showed that for most cases, both preemptive and non-preemptive techniques for verification achieve almost the same detection capability. The major drawback of the CWS system is that it always has a certain amount of error latency. However, the CWS

can be more practical than other schemes such as multiplexed caching and ECC protection because it can detect any number of bit errors, needs minimal hardware addition, uses preexisting memory redundancy in the conventional multi-level caching system, and has less impact on system performance by exploiting system bus idle time.

CHAPTER 5 On-Line Integrity Checking for Control Logic

Faults in the control logic of the microprocessor may result in incorrect execution of instructions and their sequence errors. This chapter presents a low-cost reliability enhancement strategy for the microprocessor's control logic, which usually remains unprotected against soft errors due to great overhead. We classify control signals into static and dynamic control depending on their changeability. For static control, signals used in pipeline stages are integrated into a signature and verified with a cached check code at commit time. The concept of caching signatures is introduced. The effectiveness of dynamic control is examined in which the signals are created using component-level duplication. Fault injection simulations on a SimR2K processor demonstrate that our schemes can achieve more than 99% coverage on average with a very small hardware addition.

5.1 Processor and Fault Model

Our base processor is divided into two parts. The front-end fetches instructions from cache or memory and feeds them into the back-end. In the back-end, instructions are executed and the processor's architectural state is updated. To keep the execution engine busy, the front-end may speculatively fetch instructions ahead. Depending on microarchitectural choices, the back-end can process a single or a group of instructions at a time either in or out of program order, and the number of pipeline stages and functional units varies. The processor always maintains the in-order and lookahead states correctly to support precise interrupts.

Transient faults disturb the processor circuitry used in any pipeline stage for a short period. The error may cause the system to start malfunctioning instantly or in a later cycle. An error of one type can generate new errors of other types Figure 5.1 depicts an example of such an event. A noise upsets random logic producing a control signal for storing the result of `addi` into `r2`. This fault creates an erroneous control signal, inverting *write* to *no write*, in the decode stage, and the error is latent until it causes a failure in the write stage. Since the control signal does not enable `r2` for an update, `r2` retains the old value. If `r3`+4 differs from the old value,

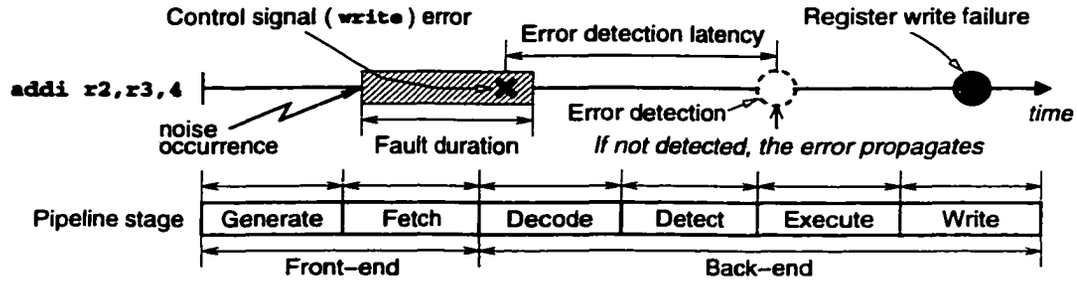


Figure 5.1 An example of control signal error propagation in a processor pipeline.

this illustrates the propagation of control error to data error. In case of reference to r2 in following operations, further propagation may be expected.

Faults in control logic result in control signal errors. In particular, errors that incorrectly change the flow of program control are referred to as control flow errors (CFEs). Control signals of the processor fall into two types. One is a signal directly derived from instructions. Each field of an instruction becomes a unique signal to control particular processor components in planned pipeline stages. This type of control for a given instruction is always the same and we call it *static control.* The other type of signals called *dynamic control* is generated by run-time conditions. Thus these signals may vary for the same instruction. The state of components and the product of several static controls determine such signals, e.g., hazard detection, bypassing, etc.

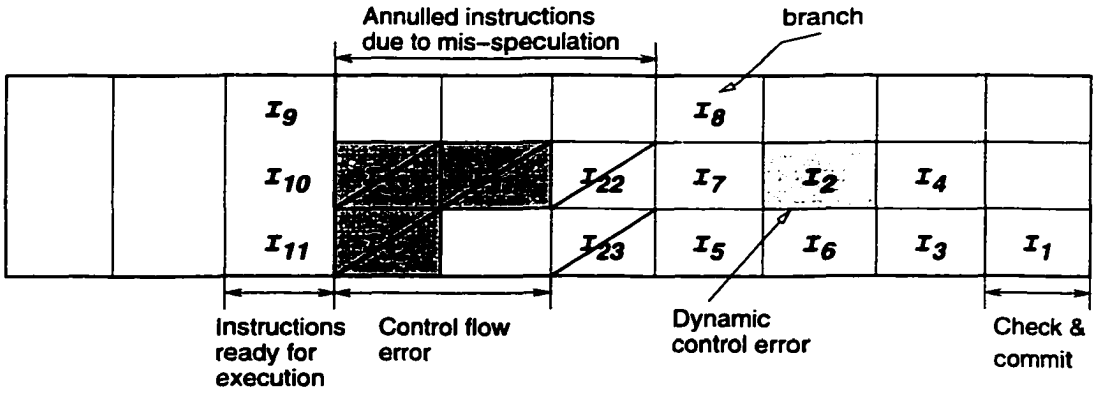
Since the circuit's susceptibility to faults vary from chip to chip and block to block within a chip, the number of circuit elements affected by a fault may vary accordingly; but a larger number could be expected for future processors [5]. The following transient fault model is used for our study. We assume that the faults occur randomly in terms of time and location. Faults cause signal inversion ($1 \rightarrow 0$ / $0 \rightarrow 1$) with an equal probability. A fault results in a k-bit error in a component, where k is dependent on the circuit's complexity and it can be as large as the output width of the component. When a fault is in action, additional faults may occur and impacts may overlap. However, the probability of occurrence of such cases is negligible.

5.2 Integrity Checking Strategy

For random logic, the common parity approach becomes ineffective since the logic output loses simple relation with the input parity. A different strategy is needed for control logic

```
I1   loop1 lui  r1, 1025          I22  loop2 ori  r7, r1, 1
I2         slt  r4, r3, r1        I23        lui  r5, 1025
I3         lui  r5, 4097          I24        sub  r9, r6, r2
I4         addi r6, r7, r8        I25        bgtz r9, loop2
I5         sb   r6, r5, 4                      •
I6         sb   r16, r5, 8                     •
I7         addi r9, r6, 8                      •
I8         bne  r1, r9, loop2     I31         add  r3, r1, r2
I9         mthi r16               I32         sb   r3, r5, 12
I10        sub  r9, r16, r6       I33         sub  r9, r6, r3
I11        beq  r1, r9, loop1     I34         bgez r9, loop1
                  •                                   •
                  •                                   •
                  •                                   •
```

(a)



(b)

Figure 5.2 Impact of control signal errors on pipelined instruction executions: (a) a sample program segment and (b) control flow deviated by a CFE in the base processor pipeline executing the example code (a).

protection. Figure 5.2 shows a sample program running on the base processor pipeline. The pipeline is about to complete I_1 and needs to ensure correct control in the earlier stages for the instruction. If there has been any error during I_1 execution, its effect needs to be removed by this point; otherwise, the system state becomes faulty.

A branch, I_8, is mispredicted as taken and instructions starting I_{22} are speculatively executed. A control error occurs in the instruction pointer multiplexer of the front-end, and then propagates causing a jump to I_{31} after I_{23}, which is not a branch instruction. This is a CFE. Fortunately, fall-through occurs in I_8 and instructions in the false flow are squashed before being committed. As a result, the effect of the CFE is eliminated as well. Errors may disappear without affecting the system state. Irrespective of the instruction scheduling, only instructions in true flow of program control are committed and need to be protected.

In this context, postponing the integrity checking till the last stage may be advantageous. This commit-time checking is suitable for the static control signals and also applicable to CFE detection if information required for verification is available at commit time.

The commit-time checking for the dynamic control is difficult to implement. For example, I_2 is data-dependent on I_1, but the bypassing unit forwards the result of I_4 instead due to a control signal error. It is costly to carry these run-time conditions to the commit stage. Our protection strategy for the dynamic control is to examine the signals on the spot with check codes when they are created. Random logic for dynamic control is duplicated at component-level or parity-protected if possible.

5.2.1 Static control protection

Unless there is an error, static control signals for a given instruction remain unchanged in every execution of the instruction. Once we know the correct signals, the integrity checking for static control logic is examining whether or not it produces and applies the same signals at the proper time. Our proposal is to transfer the signals used to the last stage in a compacted form, called *signature*, and verify the control with a pre-stored signature. The signature is an n-bit wide code ($S_0 S_1 ... S_{n-1}$) resulted from exclusive-ORing the static control signals throughout the pipeline.

Figure 5.3a shows pipelined generation of a signature. Various symbols in the boxes represent different signals that drive operations in several stages. When an instruction is fetched, its signature starts to form. The generation and use of the control signals can occur in different stages. As the instruction flows through the pipeline, signals actually used in a stage are integrated into the signature and the rest are passed to the next stage. When the instruction reaches the final stage, the signature generation is also complete. As a result, the signature manifests errors, if any, in the phases of the generation, transfer, and actual use of the static control signals.

Figure 5.3b illustrates how the static control signals used in each stage (separated by dotted lines) during an instruction execution, are integrated into a signature. Static control signals in stage i are are numbered as $C_{i,j}$, $j = 0, 1, ..., N_i - 1$, where N_i is the total number of the signal bits in stage i. The signature is basically an even parity code. For $t = 0, 1, ..., n - 1$, the signature S_t is defined by $\bigoplus_i C_{i,j}$ with $j \ mod \ n = t$ for all j, where $\bigoplus_i$ denotes the sum modulo 2 for all i. In stage i, a fault causes a 4-bit error in adjoining logic. Since those four bits affect different signature bit locations, the final signature indicates the error. A small

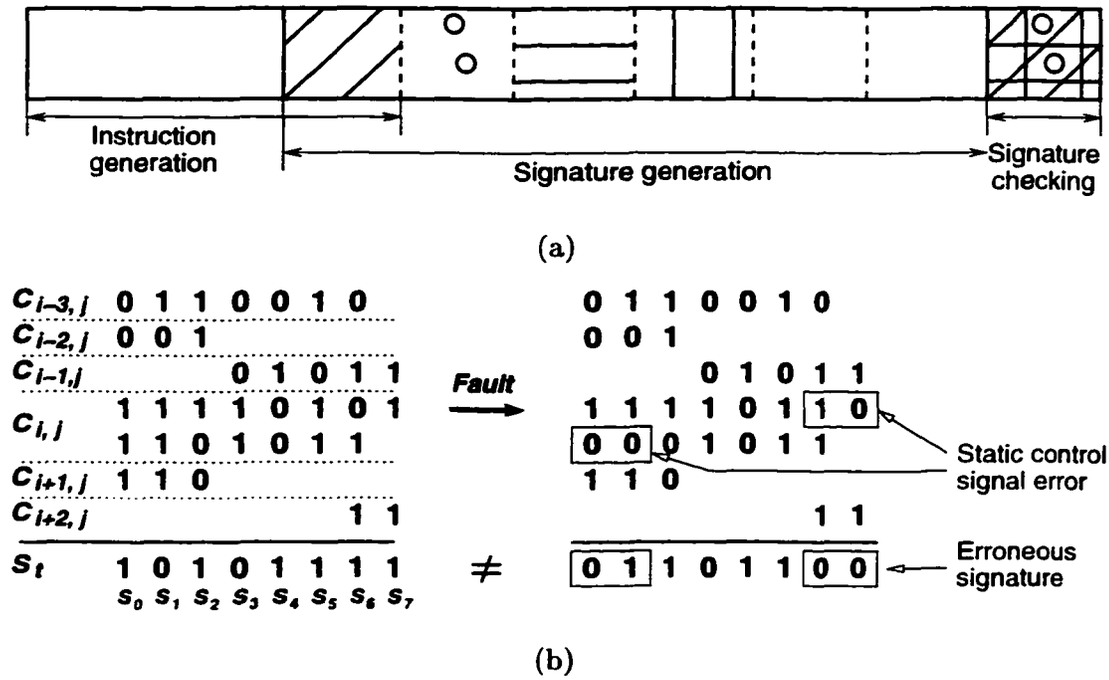


Figure 5.3 Signature of static control signals: (a) pipelined signature generation and (b) an example of signature computation with an erroneous static control signal $C_{i,j}$.

signature width n can cause error aliasing. A study on signature-based monitoring shows that 8-bit signatures are sufficient to provide high coverage [100]. We choose $n = 8$ in this chapter. If $N_i > n$, j for each signal bit position in $C_{i,j}$ is assigned in such a way that the signals from physically adjoining logic, which may be commonly affected by a single fault, are mapped to distinct values of t in generation of S_t.

In order to verify a newly generated signature at commit time, a *checking signature* needs to be provided for comparison. To maintain the checking signatures economically, we propose to generate the signatures dynamically and store them in a small table, called *signature table.* General cache management is used for the table and we call this technique *signature caching.* For simplicity, the table starts with an empty state. No initial checking signatures are prepared separately with preprocessing. On the first execution of each instruction, the run-time generated signature fills the table entry indexed by the instruction's address. This signature is used as the checking signature in later executions of the same instruction. Since the table has a limited size, it may replace old entries.

In the commit stage, the table is searched for the corresponding checking signature in

parallel. Error checking for the static control is possible only on a table hit. In case of a fault detection, i.e., signature mismatch, the pipeline is flushed and the program counter is set to the address of the committing instruction again for recovery. The execution resumes with re-fetching of the instruction. The processor already includes this kind of recovery mechanism to support speculative execution. If the checking signature is erroneous, signature mismatch persists during the recovery procedure. Unless checkpointing [3] mechanism is employed, the execution restarts from the beginning. Although signature table misses may still occur after the initial phase, high reference locality provides favorable performance of the small signature table. The proposed scheme requires no compiler modification for embedding signatures and causes no pipeline stalls for error checking.

5.2.2 Dynamic control protection

A signature of dynamic control signals can keep changing even for the same instruction. Thus, the mechanism used for the static control is not applicable. Employing a parity prediction scheme for a random logic block can cost as much as duplication. It may be possible to redesign the logic in such a way that correct output combined with some redundant output always meets a predefined condition for verification purpose. However, it is hard to guarantee a solution for all random logic, which is more advantageous than duplication in terms of error coverage and overhead.

In our strategy, we choose to duplicate the dynamic control logic at component-level. For any given unit block in the processor chip, duplication is used only for the dynamic control part of the unit that is otherwise difficult to check. For example, bypass and predicate control logic selects the output of multi-level bypass multiplexers in the integer unit. Such dynamic control logic is duplicated, while data path including the multiplexers are ECC-protected and static control signals are encoded into a signature. This minimizes chip area overhead for the integrity checking in comparison to full duplication of the integer unit. Separate checking mechanisms employed for different parts of the unit are complementary.

The two outputs of the duplicated logic are compared on every cycle. In case of a mismatch, the comparator promptly asserts error detection before erroneous signals are latched in the pipeline register. The recovery takes place without discarding any instruction in the pipeline and stops instructions from moving forward. Each pipeline stage repeats the same operation. The recovery time depends on the duration of the transient fault in the logic. The effect of a transient fault may last a few clock cycles.

In order to optimize protection capability with a limited area, a selective protection can be made based on the priority obtained from the fault sensitivity information of the logic. The fault sensitivity of logic components was discussed in Chapter 2.

5.2.3 Control flow monitoring

Errors in both the static and dynamic control logic may result in a CFE. CFEs can be classified into four types. The processor may take a jump before completing the current basic block. The jump can be taken to either the beginning (Type I) or a non-entry point of any basic block (Type II). CFEs can also occur during a branch. The processor may start to execute a new, but wrong basic block (Type III) or jump to the middle of any basic block (Type IV). Control flow monitoring (CFM) techniques detect CFEs. In traditional CFM schemes [99], [100], each basic block is associated with a signature, e.g., the checksum of the instruction stream, which is prepared before program execution. This signature is compared with the run-time generated signature at every exit point of basic blocks. Any jump from a non-exit point or to a non-entry point causes a discrepancy between two signatures, achieving error detection. However, CFEs of Type III are undetectable with this approach.

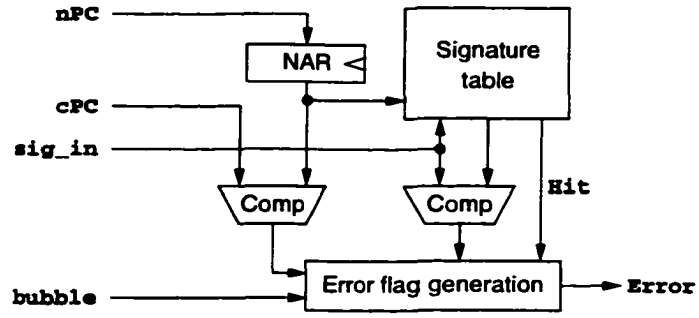


Figure 5.4 Block diagram of IA-based CFM hardware.

In our proposal, the CFM becomes a simple check, comparing the addresses of instructions being committed. Figure 5.4 shows the block diagram to implement our instruction address (IA)-based CFM scheme. The signature table is combined with the CFM hardware. When the processor executes an instruction, it maintains the address of the next instruction to be executed, denoted by nPC. Unless a branch occurs, nPC points to the next contiguous location in the current basic block. In the case of branch, nPC is updated with a branch-resolved target address in an earlier pipeline stage, and passed along with the instruction's address (cPC) and

signature (`sig_in`). In this way, `nPC` at commit time indicates the scheduled control flow. Annulling an instruction in a pipeline stage sets `bubble` to 1.

On completion of an instruction's commitment, `nPC` is stored in a register, NAR. When the next instruction is ready for commitment, its `cPC` is compared with NAR to check whether or not the instruction is from the correct flow. In case of a mismatch, i.e., CFE detection, the pipeline is flushed and the execution resumes from the instruction addressed by NAR. The signature table is also accessed with the address in NAR to check the run-time signature, `sig_in`. If `bubble` is 1, NAR is not updated and no error flag is raised. If a fault affects both `nPC` of the branch instruction and `cPC` of the next instruction (we call this case *common fault*), the fault cannot be detected. This can happen if the `nPC` and `cPC` originate from the same component. However, the shared component is protected with ECC or duplication. It should be noted that the IA-based CFM covers all the types of CFEs described above. This run-time checking can be performed outside the critical path, resulting in negligible performance penalty.

5.3 Evaluation Methodology

Fault injection simulations were conducted on a SimR2K processor that we built using Verilog HDL. SimR2K is an RTL model of MIPS R3000 32-bit CPU core. It was chosen because the architecture was generally known and auxiliary tools for design verification and benchmark generation were available in public domain. SimR2k incorporates the proposed protection schemes, and error checking takes place concurrently when a program is running. Considering scalability and RISC characteristics of SimR2K, testing the proposed strategy on SimR2K is reasonable to assess its performance in modern microprocessors. Under our fault model, component-level duplication provides 100% protection coverage for the dynamic control logic. Thus, the dynamic control protection was not separately tested.

Software simulated faults, based on the model described in Section 5.1, were injected in the control logic of the processor at run-time. For each logic component, fault injection locations (FILs) are identified after grouping fault equivalent gates together. SimR2K includes a total of 70 FILs (36 for static control logic, 17 for dynamic control logic, and 17 for data path). The complete FIL list is presented in Table 5.1 with the output of each logic block. We denote FILs by numbers and classify them by logic type and protection scheme.

Four application programs of various algorithms were used as a benchmark suite: *Hanoi*,

Table 5.1 The complete list of FILs in SimR2K: logic blocks are grouped by type and protection scheme.

FIL	Output signal		Component or operation
	Name	Width	affected by a fault in FIL
Signature caching protected			
1	zero_or_instr	32	instruction
2	extd_data	32	operand, branch
3	Br_addr	32	branch
4	Lui	1	operand, branch
5	SignBit	1	operand, branch
6	HiLoSel	1	operand
7	RegImmSel	1	operand
8	mfhi	1	operand
9	mflo	1	operand
10	RegDst	1	register data
11	Link	1	register data
12	ALUOp	3	ALU
13	WithOV	1	interrupt/halt
14	Exit	1	interrupt/halt
15	RtImmSel	1	operand
16	IDDataSt	1	memory, dependency
17	IDDataLd	1	operand
18	DataSign	1	operand
19	DataSize	2	operand
20	MemToReg	2	register data
21	RegWrt	1	register data
22	jal	1	dependency control
23	j	1	dependency control
24	jalr	1	dependency control
25	jr	1	dependency control
26	ControlIn	18	control logic
27	dst_reg	5	register data
28	HiLoWrtSel	1	HI/LO register
29	HiWrt	1	HI register
30	LoWrt	1	LO register
31	ALUCtl	4	ALU
32	MEMCtlIn	12	control logic
Signature caching and IA-based CFM protected			
33	pc_4	32	next sequential PC
34	BranchOp	3	branch
35	JumpImm	1	branch
36	JumpRs	1	branch

FIL	Output signal		Component or operation
	Name	Width	affected by a fault in FIL
IA-based CFM protected			
37	pc_in	32	PC, branch
38	IDFlush	1	instruction
39	BrFlag	1	flow control
Component-level duplicated			
40	IDRsSel	2	operand
41	IDRtSel	2	operand
42	wrt_num	5	register data
43	Stall	1	pipeline stall
44	RsSel	2	operand
45	RtSel	2	operand
46	DataInSel	1	memory data
47	MorW	1	operand
48	MulDiv	1	operand
49	ByPass	1	operand
50	ExcptFlush	1	flow control
51	PipeRegWrt	1	pipeline stall, operand
52	InstMemStall	1	pipeline stall, operand
53	DataMemStall	2	pipeline stall
ECC protected			
54	hi_lo_immd_in	32	operand
55	lo_reg_in	32	LO register
56	hi_reg_in	32	HI register
57	hi_or_lo	32	operand
58	reg_rs_out	32	operand
59	reg_rt_out	32	operand
60	rt_data_in	32	operand
61	Jr_addr	32	operand, branch
62	lo_data_in	32	LO regsiter
63	hi_data_in	32	HI register
64	st_data	32	memory data
65	rs_alu_in	32	operand
66	rt_alu_in	32	operand
67	immd_bypass	32	operand
68	data_in	32	memory data
69	WB_reg_data	32	register
70	ld_data	32	operand

Static control logic

Dynamic control logic

Data path

Intmm, *Heap*, and *Queens*. All benchmarks were written in C and compiled with a cross-compiler, `dlxcc` [101]. When a benchmark program runs on SimR2K, a single fault injection is performed into a targeted FIL at a randomly selected clock denoted by fault injection point (FIP) for one clock cycle. Multiple FIPs are examined for each FIL independently, and the number of FIPs is determined by the experiment time limit and desired accuracy of the measurement.

Table 5.2 Outcome classification of fault inject run.

Case	Error	Program termination	Result & system state comparison	Effect of error
1	Not detected, effective	at T	mismatch	*Program failure*
2		before T		
3		at T+500		
4		after T, before T+500		
5	Not detected, but not effective	before T	match	None or negligible
6		after T		
7		at T		
8	Detected by IA-based CFM	after T		Easy to recover
9	Detected by signature caching			
10				Hard to recover

On completion of each benchmark run in the presence of a fault, the outcome is evaluated by a comparison with the precomputed correct results and final architectural state. Table 5.2 categorizes possible outcomes of the benchmark run with a fault injection into 10 cases. We assume that program always fails after T+500 cycles, where T is the normal execution time of each program. A continuous run is forcibly stopped at T+500. If a fault is not detected (Case 1~7), a failure may or may not occur. Undetected faults that cause the program execution to fail are our major concern. Therefore, we measure the performance of the proposed schemes with a fault protection *coverage* computed by (*number of successful runs/number of fault injection runs*)$\times$ $100\% = (1 - \sum_{i=1}^{4} Case\ i / \sum_{j=1}^{10} Case\ j) \times 100\%$.

5.4 Experiment Results

The combination of the signature caching and the IA-based CFM covers 39 FILs (36 static and 3 dynamic control logic blocks) as shown in Table 5.1. The redundancy added for that is only a 512-byte table (2-way set associative cache with 64 signature entries of 8 bytes) and

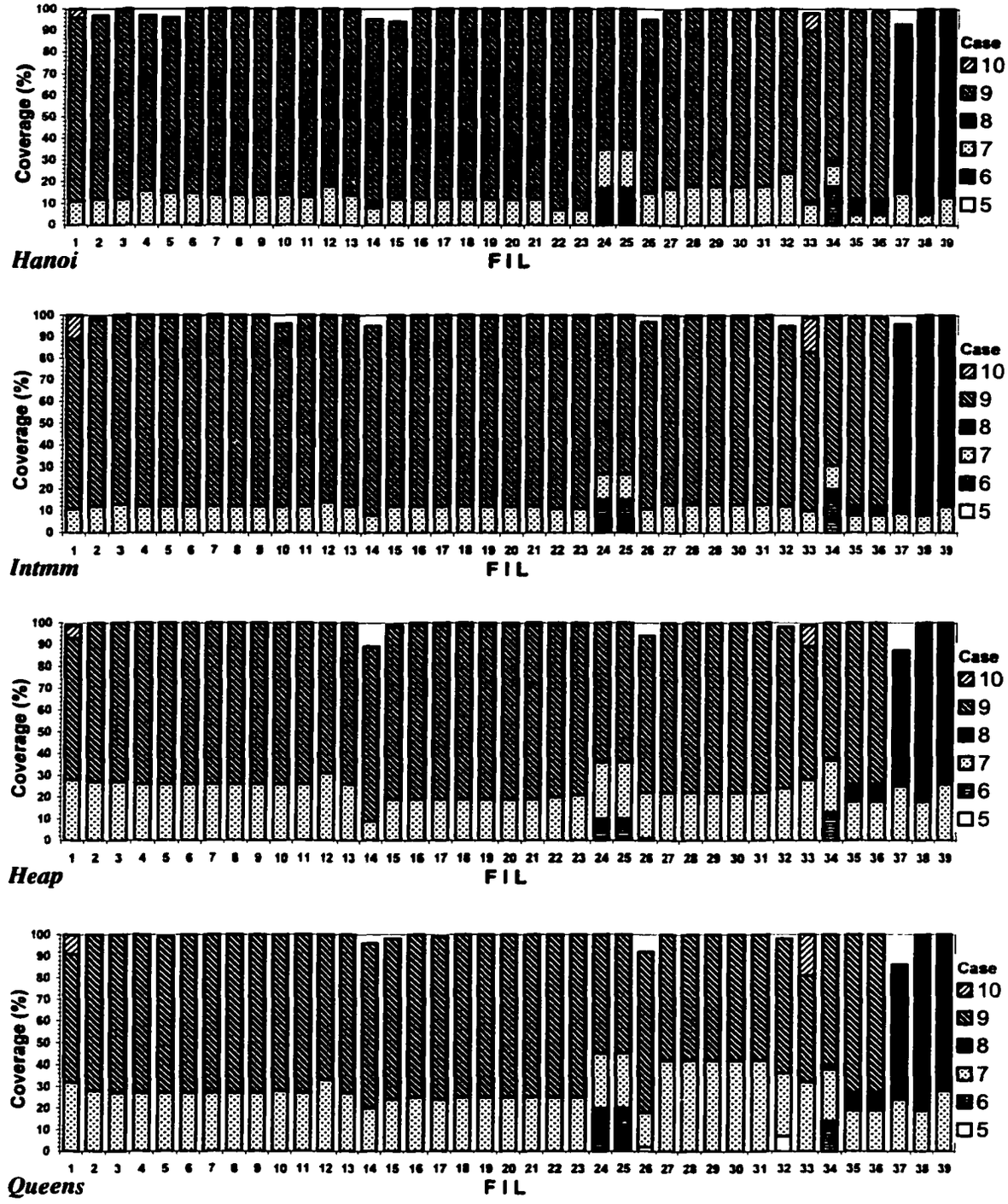


Figure 5.5 Fault coverage for the benchmarks: outcome distribution also shows error detection coverage.

small supporting logic. The remaining 14 FILs of dynamic control are checked with component-level duplication. The area of control logic blocks that can be protected by each technique changes for different processor architectures. However, compared to the unit- or system-level duplication, the hardware overhead is very minimal. For SimR2K, we expect roughly 60% reduction in gate counts.

The coverage of the signature caching can be affected by the processor's run-time behavior. The signature caching cannot perform integrity checking in the case of a table miss. To make its small hardware addition meaningful, its fault coverage needs to be considerably high. Figure 5.5 presents the performance of the signature caching collaborating with the IA-based CFM for four benchmarks. For each FIL on x-axis, the coverage shown was obtained for faults at 100 FIPs. As clearly seen from the figure, most signals are completely covered. On average, no more than 1% of injected faults result in a failure for all benchmarks. This is mainly the result of two factors: 1) locality in instruction execution is generally high and 2) not every undetected fault generates a critical error, disturbing program execution. Possible error of this coverage estimate is less than 7% in all cases for a 95% confidence level.

The distribution of fault injection outcomes, based on Table 5.2's classification, is also shown. Interestingly, benchmarks have common general tendencies in the distribution. This indicates that the characteristics of control logic are more important than the program being executed with respect to the effect of faults. Table misses are responsible for faults that are not covered in the signature caching. The coverage loss of the IA-based CFM in FIL 37 corresponds to the common fault cases described in Section 5.2.3. This appears because ECC protection for the common source was not included in the simulation.

Faults missed by the signature checking may be captured by the CFM, and vice versa. The reason that Case 9 is always dominant over Case 8 is that any case where the signature caching and the CFM detect the same error was counted as the coverage of the signature caching. Most of Case 10, i.e., error checking with an incorrect signature in the table, can be observed in FILs 1 and 33. Therefore, once an error is detected by our mechanisms, the recovery is usually simple.

5.5 Summary

We have presented a comprehensive integrity checking strategy for the control logic of microprocessors, which is a hard-to-protect area. The scheme is devised after examining the

effects of faults on the processor and the characteristics of the control logic. Control signals are classified into static and dynamic control and separate protection approaches are applied. We have introduced the concept of caching signatures for the static control protection. By exploiting locality in program code, this technique significantly reduces the overhead required in traditional signature-based checking while providing high protection capability. In addition, commit-time checking eliminates unnecessary checking in earlier stages and allows us to use the existing mechanism for recovery. Our IA-based CFM technique can detect all four types of CFEs with the assistance of the signature caching whereas conventional methods cover three types only. For the dynamic control protection, we change the replication from a large unit-level to a minimal component-level. This can take advantage of selective redundancy allocation based on the fault sensitivity of the control logic.

The proposed techniques simultaneously achieve our goals of 1) low-cost, 2) high fault coverage, and 3) easy recovery. We expect that its performance penalty is small. Our schemes and the current techniques are complementary to the full integrity checking of microprocessors. Hence, incorporating the proposed techniques will significantly enhance the processor's dependability.

CHAPTER 6 System Level Fault Tolerance for Superscalar Processors

This chapter proposes an integrity checking architecture for superscalar processors that can achieve fault tolerance capability of a duplex system at much less cost than the traditional duplication approach. The pipeline of the CPU core (P-pipeline) is combined in series with another pipeline (V-pipeline), which re-executes instructions processed in the P-pipeline. Operations in the two pipelines are compared and any mismatch triggers recovery process. The V-pipeline design is based on replication of the P-pipeline, and minimized in size and functionality by taking advantage of control flow and data dependency resolved in the P-pipeline. Idle cycles propagated from the P-pipeline become extra time for the V-pipeline to keep up with program re-execution. For a large-scale superscalar processor, the proposed architecture can bring up to 61.4% reduction in die area and the average execution time increase is 0.3%.

6.1 Virtual Duplex System

For high-end servers that require high reliability and availability, replication at unit- or system-level is often considered to be the ultimate solution for fault tolerant processor design [103], [104]. This approach can provide 100% error detection without introducing much design complexity. A major drawback of the system-level replication is high additional cost, and this usually restricts its use. Even if a sufficient die area can be budgeted for multiple processor modules, power dissipation issue may be an obstacle in some applications. To cope with the increasing reliability concern for a broader range of microprocessors, it is desirable to develop more affordable integrity checking mechanisms with good fault coverage and performance characteristics.

This chapter presents an inexpensive fault tolerant architecture for modern superscalar processors. The main idea is to exploit idle cycles in the processing core for its integrity checking with a serial verification core. When instructions are ready to be committed in the original core, the verification core re-executes them to detect any errors. Since two cores are connected in series rather than in parallel, a significant hardware optimization in size and

functionality is achievable for the verification core. This is called a selective series duplex (SSD) architecture. We develop our proposal by suitably modifying existing architectures with the following goals in mind: 1) to provide the protection capability of a duplex system, 2) to minimize hardware requirement for duplication, 3) to maintain the maximum performance of the original core, and 4) to keep design and verification complexity very low.

6.2 Fault Manifestation in Microprocessors

Soft errors manifest temporary environmental disturbances such as power jitter, electromagnetic interference, cosmic rays, etc. Figure 6.1 illustrates an example of the transient fault occurrence in a small logic block of a processor chip. A neutron-hit creates a burst of electronic charge in the circuitry of a NAND gate, which changes the gate output from 1 to 0. Continuous correct input signals make the gate recover at the output since no irreversible physical damage has been done. This recovery may take several clock cycles as the amount of the noise burst increases. If the output is latched before the recovery, it may propagate to other components and eventually cause a computation failure. If cosmic rays hit a memory cell, its logic value may be flipped and remain erroneous until new value overwrites it.

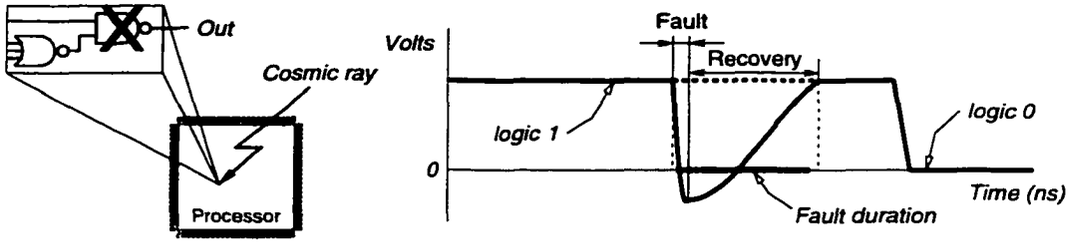


Figure 6.1 Transient fault manifested in a logic output of a processor chip.

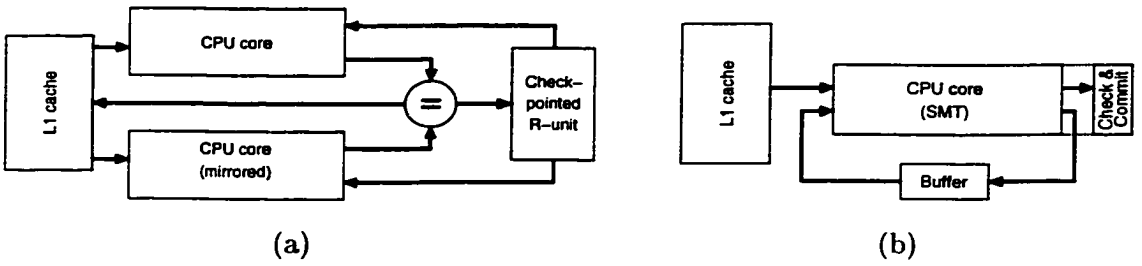
Fault behavior is random. Faults can occur in any component of the processor at any point of operation and they may affect the circuit very shortly or permanently in diverse ways. The circuit's susceptibility to the faults is not the same even within a logic block. Accordingly, the number of components affected by a single fault changes. Moreover, the situation may vary with the systems and operating conditions. It is impractical to make a processor completely tolerant to all such randomness of faults. Instead, we can define a fault model based on the likelihood of different cases and develop a system under that fault model. The first assumption is that a fault in a component only appears as the inverted output signal, high to low or low

to high with an equal probability. Also, a fault can corrupt multiple bit signals simultaneously and more than one such faults may be active at the same time. We assume that the probability of fault occurrence is uniformly distributed over time and location.

Duplication is one of the simplest approaches that can match up to this fault model. Since effective faults are manifested in outputs, without considering intermediate processes only final outputs need to be checked and the comparison is a very efficient method for that. Two operations and results that are replicated at the system-level eliminate any limit on detectable number and types of faults unless a failure is identical in both modules. Duplication evenly provides fault tolerance capability for every component. However, it is not effective in the case of common failures. The comparison element is a single point of failure. Thus, it needs to include a totally self-checking mechanism [105]. Physical division is preferred to reduce the probability that a single fault results in a common failure. Independent faults in two modules may produce identical faulty outputs, but the probability of occurrence of such events is expected to be very low. All architectures discussed in this chapter adopt a form of replication, and therefore the same fault criteria should be used to assess their fault coverage.

6.3 SSD: Selective Series Duplex Architecture

We apply hardware replication with a different organization to combine the benefits of existing fault tolerant designs into the SSD architecture. Figure 6.2a shows the organization of the IBM S/390 G5 microprocessor [103], in which the CPU core is duplicated in a conventional way. The two CPU cores are configured in parallel and operate in lock-step to compare the outputs in every cycle before going to the L1 cache and the register unit (R-unit), which are shared and ECC-protected. This architecture is robust in error detection and easy to implement. However, it utilizes resources inefficiently. Both cores always perform exactly the same operations irrespective of their usefulness. For example, if a branch misprediction occurs in one core, the identical misprediction is also made in its counterpart. This is not to verify the prediction, but to keep the two cores synchronized. In fact, faults in prediction logic, causing a misprediction, do no harm to the execution integrity. If the pipeline in one core generates pipeline stalls, the other pipeline produces the stalls at the same cycles. A lot of hardware resources and CPU cycles are wasted just for design simplicity. It should be noted that operations involved in executing instructions that are not committed, i.e., not in the real control flow, need not be checked.



(a) (b)

Figure 6.2 (a) A conventional fault tolerant processor with a dual CPU core. (b) Fault tolerance using re-execution through a single multithreading processor core.

Instead of duplicating the processor, integrity checking can be accomplished by executing each program twice through a single core. If multithreading is used, the primary and secondary executions can be overlapped [106], [107], [108]. Figure 6.2b depicts an organization for this approach. Each program and its copy are treated as independent threads and executed in parallel. Only completed instructions during the primary execution are sent to the buffer along with the results. The secondary execution is performed by taking instruction entries from the buffer that may also include the control and data flow information of the primary execution. Utilizing such information can make the secondary execution simpler. The processor may make use of idle cycles of one thread for another thread to reduce execution time. Although this dynamic re-execution architecture achieves fault tolerance, the program execution time still increases by 5~30% even with a significant addition of execution engine. Since a single pipeline is shared with both threads, the actual optimization for the secondary execution is less than expected, and thus re-execution time cannot be completely absorbed away. One important lesson from here is that the redundant execution overhead can be minimized by utilizing idle resources. In [28], augmenting the commit phase of the processor pipeline with a checker was proposed for functional verification in microprocessor designs. This approach however uses a heterogeneous checker that may require large design and verification overhead. Besides, the checker is unrealistically assumed to be fault free.

Efficient integration of redundant execution into the pipeline is a key issue to realize inexpensive fault tolerance. Our proposal is to *duplicate* the pipeline for the re-execution in a *selective* way and configure it in *series*. Figure 6.3 illustrates the proposed organization where the original CPU core, *P-pipeline*, is combined with an optimized duplicate, *V-pipeline*, in sequence, forming one long pipeline. The P-pipeline is a conventional superscalar processor core with precise interrupt handling. The verification core of the V-pipeline is created from the

P-pipeline after removing unnecessary front-end logic including branch prediction and prefetch hardware, IP generation MUXes, etc. To keep the complexity minimum, the V-pipeline reuses the P-pipeline design as much as possible. Furthermore, the V-pipeline is down-scaled throughout all the processing units such as buffer, decoder, ALUs, and so on. A minimal hardware addition is required to support efficient re-execution and error checking.

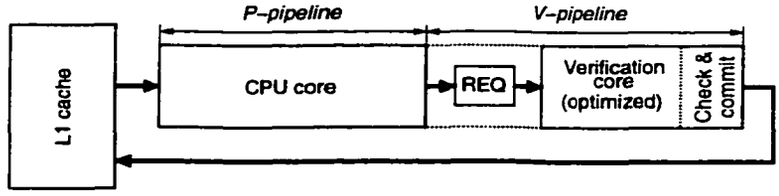


Figure 6.3 Selective series duplication (SSD).

A FIFO type buffer, called *re-execution queue* (REQ), is inserted between the two processing cores. When an instruction is about to be committed in the P-pipeline, the REQ takes the instruction along with its result and provides them to the verification core. Although the P-pipeline can complete instructions in out of program order, instructions are buffered in commit order, which makes the hardware simple and maintains an instruction stream with free of control flow hazard. The execution results in the REQ are also used to immediately resolve data dependency, if any, in the decode stage of the V-pipeline. If the REQ is full, it cannot receive any new instructions and the commitment in the P-pipeline is delayed. This is the case where the re-execution may increase the execution time. Therefore, the V-pipeline should be able to process queued instructions fast enough to keep room available in the REQ.

Each load or store instruction requires two operations, effective address calculation and cache or memory access. Extra memory ports for the verification purpose may be too expensive. The SSD architecture requires no additional memory ports. Effective address is checked by means of dual computations like other instructions and actual memory access occurs only once for each memory instruction. The P-pipeline loads data into its registers as usual, while it stores data into a store buffer instead of the memory. On the other hand, the V-pipeline does not access the memory for a load, but it finally stores data into the memory if the data item stored in the store buffer by the P-pipeline matches with the data being stored by the V-pipeline. As a result, real load and store memory accesses are performed in the P-pipeline and V-pipeline, respectively. Because data and instructions in the memory are protected with an ECC, errors are checked on every read access.

Fault detection is achieved by executing each instruction twice consecutively, and the two results are compared before committed to the architectural state in the V-pipeline, i.e., *commit time checking*. This means that the V-pipeline always maintains a correct check-pointed processor state. The commit time checking prevents errors from propagating to the processor state. Although the two pipelines are not identical in size, they produce the same results under the fault free condition. If a fault is detected, both pipelines are flushed and a subroutine call carries out a rollback recovery.

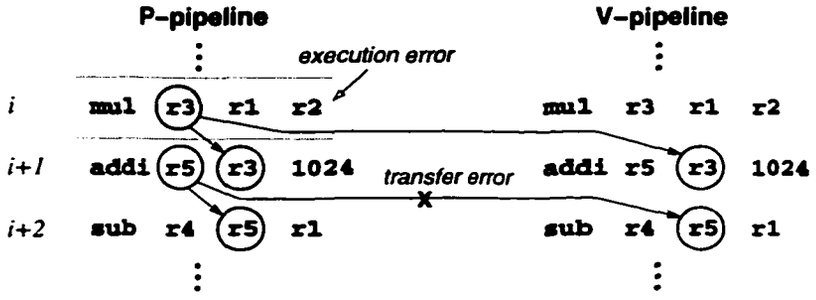


Figure 6.4 An example of instruction execution and error detection in the SSD pipeline.

Figure 6.4 shows a small program segment being executed in two pipelines of a SSD processor. During the primary execution in the P-pipeline, because of data dependency on r3 and r5, instructions $i+1$ and $i+2$ should wait until instruction i of a long latency is completed. On the other hand, in the case of the V-pipeline instructions $i+1$ and $i+2$ can be executed immediately, since r3 and r5 are transferred from the P-pipeline via the REQ. If the execution of instruction i in the P-pipeline is erroneous, the subsequent instructions in the both pipelines become incorrect. However, the re-execution of instruction i in the V-pipeline detects the error before the other instructions are committed. A faulty data transfer, i.e., r5 corruption in an REQ entry, is detected by the result comparison of instruction $i+2$ in the V-pipeline.

The characteristics of the proposed SSD architecture are as follows. The V-pipeline needs to execute only instructions in the actual flow of program control that are passed from the P-pipeline for verification. Since the V-pipeline never executes instructions speculatively, it requires no speculation hardware and processes fewer instructions than the P-pipeline. The V-pipeline takes advantage of the dependency hints and execution results pre-resolved and transferred from the P-pipeline to avoid dependency-induced pipeline delays. The P-pipeline can hardly sustain its maximum throughput and its stall cycles propagate to the V-pipeline.

Exploiting this idle cycle makes it possible for the V-pipeline to have fewer functional units than the P-pipeline without degrading performance. The V-pipeline may start re-execution as soon as a completed instruction is ready for commitment from the P-pipeline, and the execution may be in out of program order for a speed up. The multithreading support is not necessary for the re-execution. The latency for each instruction's commitment increases due to expanded pipeline, but on the whole it is only a small initial delay and operations are mostly overlapped. The penalty for a misspeculation remains the same because the speculative execution is still verified in a P-pipeline stage, causing no impact on instructions being executed in the V-pipeline.

6.4 Evaluation Environment

We have modeled the SSD architecture in simulation software and examined its performance impact and hardware cost compared with non-redundant baseline processors. The SSD processor simulator was developed from the simplescalar tool set [109] mostly by replicating necessary processor blocks for the V-pipeline. The baseline processor is configured for high throughput with wide-issue, out-of-order superscalar, and branch prediction.

Table 6.1 lists standard parameters used in our evaluation. The baseline processor includes only one CPU core (P-pipeline) and memory system. The SSD processor comprises the baseline and a V-pipeline, which is a down-scaled version of the P-pipeline. To test the SSD architecture for a range of superscalar processors, we fixed the baseline configuration into three classes, *large-scale*, *mid-scale*, and *small-scale*, while varying the scales of the V-pipeline for each baseline class. The table enumerates the V-pipeline parameter values examined. Optimal configurations of the V-pipeline could be found through extensive simulations. For the sake of presentation, each configuration of the V-pipeline is denoted by a sequence of parameter values in the order of REQ size (q), decode/issue/commit bandwidth (d/i/c), register renaming/updating unit (r), load/store queue (l), and functional units. For example, 8-4/4/4-64:16-4/1/3/1 denotes a V-pipeline with the same configuration of the *small-scale* P-pipeline described in Table 6.1.

For each selected SSD configuration, the processing speed of benchmark programs was measured by execution-driven simulations and the corresponding chip area overhead was estimated. We used seven SPEC95 programs, pisa big-end binaries obtained from [110]. The simulator was compiled on a Sun Ultra-10 under SunOS 5.7. We ran each benchmark on the simulator for 200 million instructions with inputs shown in Table 6.2. The performance metric

Table 6.1 The baseline and SSD processor configuration.

Parameters		*Baseline CPU core (P-pipeline)*			*V-pipeline*
		large-scale	*mid-scale*	*small-scale*	*Various scales*
Instruction fetch queue (instr. entry)		16	16	8	Re-execution queue (REQ): 2, 4, 8, 16
Decode / issue / commit bandwidth (instructions/cycle)		8 / 16 / 16	4 / 8 / 8	4 / 4 / 4	1,2,4,8 / 1,2,4,8 / 1,2,4,8
Reorder buffer (instr. entry)		256	128	64	4, 8, 16, 32, 64, 128
Load/store queue (instr. entry)		64	32	16	2, 4, 8, 16, 32
Functional units	int ALU	6	4	4	1, 2, 3, 4, 5
	int MUL/DIV	2	2	1	1
	fp ALU	6	4	3	1, 2, 3, 4
	fp MUL/DIV/SQRT	2	1	1	1
Branch mis-prediction penalty (cycles)		8			no penalty
Branch prediction		gshare with 2048 entries 16-bit global history			no prediction needed

Memory hierarchy	
L1 instruction cache	64KB, 2-way, LRU, 8 instructions/block, hit/miss=2/12 cycles
L1 data cache	64KB, 4-way, LRU, 32B/block, hit/miss=2/12 cycles, 2 read/write ports
L2 unified cache	1MB, 4-way, LRU, 64B/block, hit/miss=10/60 cycles

of our interest is *relative execution time* (RET) that is the total execution cycle ratio of the SSD processor to the baseline processor for a given program.

Besides presenting only simulated values for the parameters in Table 6.1, we also make an estimation of the die area using an area breakdown shown in Table 6.3. This provides us better insight into the area requirement for the SSD. The area factors are determined on the basis of the die area distribution data for four commercial microprocessors [111], [112], [11], [113]. The area factor is the relative die area of unit functional block with respect to a single entry of the instruction fetch queue. For example, the area factor of a 4-instruction wide decoder is 99.9. Both the P-pipeline and V-pipeline include separate 128 integer and 80 floating-point

Table 6.2 Benchmarks and input files.

Benchmark	go	li	perl	hydro2d	swim	tomcatv	wave5
Input file	9stone21.in	test.lsp	primes	ref	train	train	test

Table 6.3 Die area breakdown.

Functional block	Area factor	Baseline processor area estimate		
		large-scale	mid-scale	small-scale
Instruction fetch queue	1.0	16.0	16.0	8.0
Re-execution queue (REQ)	2.0	0.0	0.0	0.0
Instruction decode	25.0	199.8	99.9	99.9
Instruction issue	25.0	399.6	199.8	99.9
Instruction commit	5.0	80.0	40.0	20.0
Register renaming/update	2.7	703.7	351.9	175.9
Load/store queue	2.7	175.9	88.0	44.0
Integer ALU	29.3	175.5	117.0	117.0
Integer MUL/DIV	35.1	70.2	70.2	35.1
Floating point ALU	56.4	338.3	225.6	169.2
Floating point MUL/DIV/SQRT	99.7	199.3	99.7	99.7
Integer register file	0.5	64.0	64.0	64.0
Floating-point register file	0.7	57.6	57.6	57.6
Speculation unit	70.3	70.3	70.3	70.3
Total		2550.3	1499.9	1060.5

registers. A common area factor of the speculation unit is used for all configurations. The table lists total area factors for three baseline processors. The additional *area requirement* for a SSD processor is the total area factor of the V-pipeline divided by the total area factor of the P-pipeline. Here the requirement is considered in comparison to a traditional duplex system.

6.5 Cost Reduction and Performance Impact

Two important goals of the SSD architecture are substantial cost reduction in building a duplex system and negligible performance overhead. This section evaluates the architecture in those terms with the simulation results. Figure 6.5a shows the RETs of the large-scale SSD processors, in which the *large-scale* baseline is checked by the V-pipelines of different sizes. The line graph denotes the area requirement for the V-pipelines scaling up from 21.7% to 61.7%. When it becomes 38.6%, the average RET is 1.003. Further up-scaling to 51.4% reduces the maximum observed RET to near 1 (0.9% execution time increase for *swim*). These results indicate that a V-pipeline with only 38.6% area of the P-pipeline is capable of re-executing all the instructions retired from the P-pipeline without slowing down the processor much. This happens because the number of instructions that the V-pipeline needs to execute is usually less, whereas the P-pipeline may consume its resources in processing incorrectly speculated instructions. More importantly, the P-pipeline can be stalled by control and data hazards, but that is not the case in the V-pipeline. As a result, the re-execution is faster than the primary

execution and requires less hardware resources. The P-pipeline delays virtually correspond to extra cycles that the smaller-sized V-pipeline may take to keep pace with it.

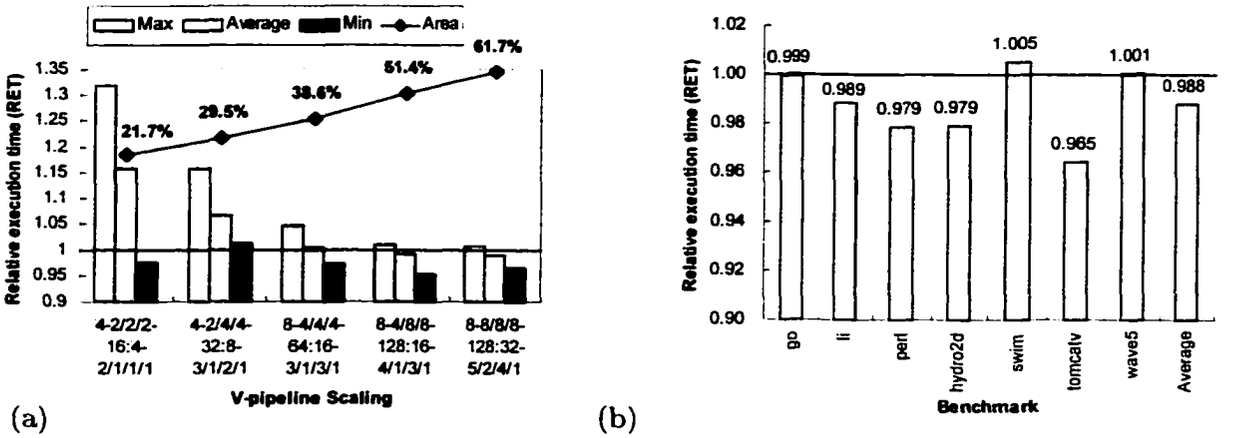


Figure 6.5 The RETs of the large-scale SSD system: (a) Effect of scaling V-pipeline. (b) RET for each benchmark (V-pipeline: 8-8/8/8-128:32-5/2/4/1 of 61.7% area).

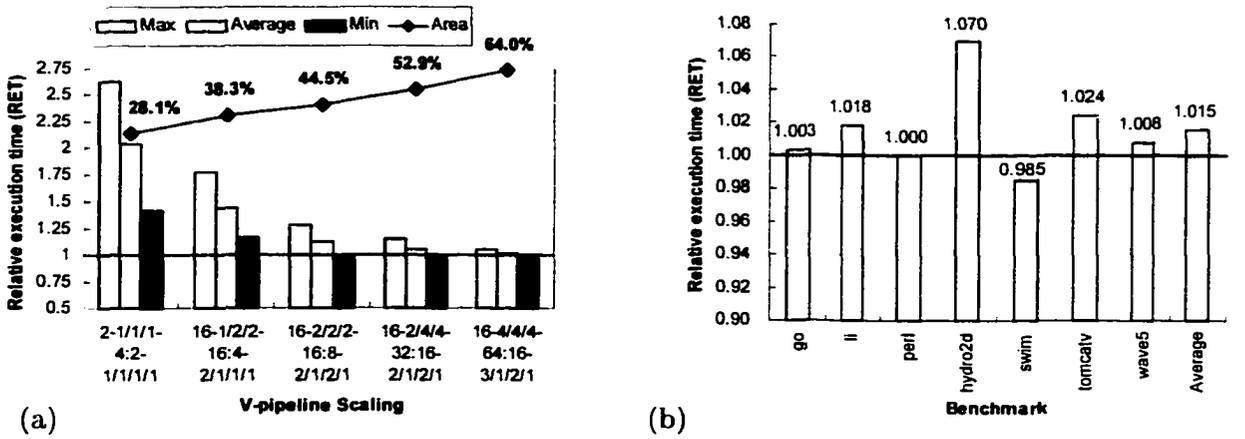


Figure 6.6 The RETs of the mid-scale SSD system: (a) Scaling V-pipeline. (b) RET for each benchmark (V-pipeline: 8-4/4/4-64:16-3/1/2/1 of 61.9% area. Note this case is not shown in (a)).

We have found that increasing REQ size from 4 to 64 can bring an average 0.9% RET reduction. However, in general increasing the V-pipeline on the whole with a balance is more effective. Figure 6.5b presents the RET for each benchmark on a SSD processor with a V-pipeline of 61.7% area. Interestingly, for some benchmarks execution time on the SSD is less than that on the baseline processor. The average RET is less than 1, and 3.5% performance

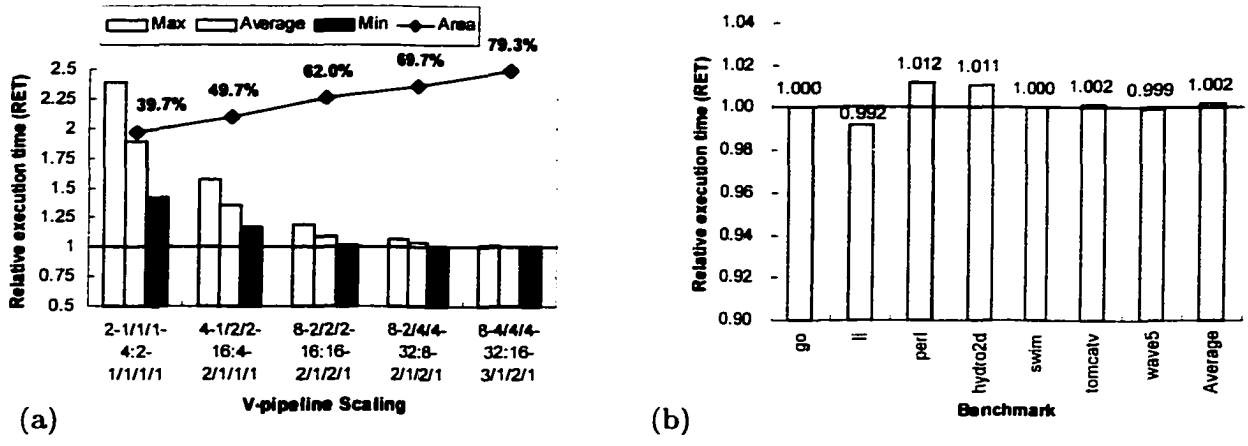


Figure 6.7 The RETs of the small-scale SSD system: (a) Scaling V-pipeline. (b) RET for each benchmark (V-pipeline: 8-2/4/4-32:8-2/1/2/1 of 69.7% area).

improvement is observed in the case of *tomcatv*. The reason for this is that additional buffering for store provided by the V-pipeline helps to avoid slowing down the processor when all memory ports are busy. In the baseline processor, if a store cannot be committed due to lack of memory port, the commitment of the subsequent instructions is also delayed even if they are ready. On the other hand, store accesses are immediately passed to the V-pipeline in the SSD unless the REQ is full. While actual store accesses are buffered in the load/store queue of the V-pipeline, memory ports may become available and stores are serviced at commit time in the V-pipeline. It should be noted that the frequency of such a case is affected by the benchmark nature and other V-pipeline parameters as shown in Figure 6.5a: the minimum RET fluctuates in spite of increase in the load/store queue size.

The overall resource utilization of the large-scale baseline processor might be lower than that of smaller-scale baselines, but the greater hardware resources can exploit more instruction level parallelism, resulting in a higher throughput. Our claim here is that a lot of redundant resources in a superscalar can be excluded when the processor is replicated for reliability enhancement. Figure 6.6 shows RETs of the SSD system built on the *mid-scale* baseline. The SSD approach still achieves the goals: only 64% area of the mid-size P-pipeline is enough to implement a V-pipeline that can maintain the average RET of 1.013. When a V-pipeline of 61.9% area is applied (Figure 6.6b), the performance improves a little for *swim*, and the average RET is 1.015. The simulation results of the SSD with the *small-scale* baseline are shown in

Figure 6.7. In this case, it takes 79.3% area to obtain the average RET of 1.002. As seen from the graphs, relative area requirement for the SSD increases for the smaller scale baseline processors. This is due to decreasing redundant resources in the smaller baselines at the cost of lowered performance. However, the actual size of an optimal V-pipeline for desirable performance is in fact reduced by 7% and 17% for the mid-scale and small-scale baseline, respectively. This is expected because the baseline of a decreased throughput requires less processing capability of the V-pipeline.

6.6 Summary

Low cost reliability enhancement techniques are needed to deal with increasing fault rate in deep submicron microprocessors. In this chapter, we have developed a new fault tolerant architecture for modern superscalar processors, SSD, which economically realizes a virtual duplex system for increased dependability. The SSD architecture bases hardware duplication approach to exploit its robustness in integrity checking. The underlying idea is to organize the original CPU core, P-pipeline, and its verification counterpart, V-pipeline, in series instead of in parallel. This allows the V-pipeline to take advantage of the idle cycles and pre-execution results from the P-pipeline in its fast re-execution for error checking. As a result, hardware resources required for the V-pipeline can be minimized.

We have demonstrated that our SSD approach can effectively achieve the following goals.

- *Duplex-system-like fault coverage.* Since operations are performed twice through two pipelines for the instructions in the actual flow of program control, it can detect any soft errors as well as permanent faults unless they are in a common failure mode. Besides, the commit time error detection makes it possible to adopt existing misspeculation recovery mechanism as error recovery.

- *Considerable hardware cost reduction.* We have obtained optimal V-pipeline configurations with an average RET of near 1 for three sampled classes of baseline processor. Estimated area reduction is 61.4% for the large-scale, 36.0% for the mid-scale, and 20.7% for the small-scale baseline in building a SSD processor. The scalability of the SSD architecture also enables the trade-off between performance and reliability depending on chip area budgeted.

- *Negligible performance overhead.* Our cycle accurate simulation results show that RETs

quickly decrease as the V-pipeline is scaled up. The average execution time increase incurred by the above optimal V-pipelines is only 0.3%, 1.3%, and 0.2%, respectively.

- *Low design and verification efforts.* The V-pipeline is built mostly with the same design of the P-pipeline functional block, minimizing design overhead. This re-usability also applies in the test sense.

Considering presented advantages, we believe that the SSD architecture is worth considering in designing future fault tolerant microprocessors.

CHAPTER 7 Future Work

We will continue to focus on cost-efficient and flexible fault tolerance schemes. Our future work includes extended measurement and analysis of the fault sensitivity using various classes of processor chips. Criteria to analyze the fault sensitivity of different logic blocks will be further elaborated. We will investigate the sensitivity data at multiple levels of design hierarchy and develop low-overhead methods for fault sensitivity prediction. If the sensitivity is easily predictable, time consuming fault simulations are not required. We will also verify the practicability of sensitivity deduction from the processor architecture. We will develop a comprehensive guideline for redundancy allocation with which the designers can intelligently optimize protection capability.

In order to fully validate the benefit of the schemes presented in this dissertation, the geometry of physical chip area in the VLSI design needs to be investigated. We will completely implement our architectures in the RTL model and synthesize to measure exact area effectiveness of the techniques for different major processor models. The impact on the processor's normal operations will also be examined through simulations. We will conduct fault injection experiments and estimate the fault coverage for more diverse fault models to include all possible scenarios in the actual system.

For system-level approaches, we will explore *system on chip* practices and investigate reusability in design and verification. In addition to scaling, other optimizations for faster re-execution will be a part of our future work.

CHAPTER 8 Concluding Remarks

In this dissertation, a study has been conducted to understand the microprocessor's sensitivity to transient faults and develop low-cost fault tolerance techniques. The proposed run-time integrity checking and error recovery may make it possible to use more chips in a wafer of a low yield with fewer testing and validation processes. This eventually enables the microprocessor manufacturers to ship more chips per unit time.

Our fault injection study on the picoJava-II and SimR2K processor demonstrates that there are many functional blocks with very low sensitivities. The fault sensitivity is determined by architectural functions, logic characteristics, and active cycles. If we exploit the fault sensitivity information, cost-effective dependability enhancement may be easily achievable.

For memory protection, we have proposed parity caching, shadow checking, and selective checking mechanisms for the situations where enough area cannot be budgeted to support the conventional uniform organization of the protection codes. We have shown that our locality-based configuration schemes for the check codes have a great flexibility of area utilization in optimizing protection capability. We have also developed a new cache write error detection technique in multi-level caching systems. Our CWS scheme provides almost complete coverage for the I-cache without interfering with the normal memory access.

For control logic, a hard-to-protect area of the microprocessors, we have presented a comprehensive integrity checking strategy. Our schemes take advantage of unique characteristics of the control logic. The signature monitoring and information caching are combined to exploit locality in program code. This technique significantly reduces the overhead required in traditional signature-based checking and provides more than 99% protection coverage. Our schemes and the current techniques are complementary to the full integrity checking of the control logic.

Finally, our SSD architecture for modern superscalar processors has shown that realizing a virtual duplex system is still possible with much less hardware resources compared to the conventional duplication. Estimated area reductions are 61.4%, 36.0%, and 20.7% for large-

scale, mid-scale, and small-scale baseline processors, respectively. The corresponding average execution time increases are 0.3%, 1.3%, and 0.2%. This advantage is obtained from a simple modification, from parallel to series, in pipeline organization.

BIBLIOGRAPHY

[1] J. Karlsson et al., "Using heavy-ion radiation to validate fault-handling mechanisms," *IEEE Micro*, 14(1):8-23, Feb. 1994.

[2] J. Sosnowski, "Transient fault tolerance in digital systems," *IEEE Micro*, 14(1):24-35, Feb. 1994.

[3] D. Siewiorek, R. Swartz, *Reliable Computer Systems: Design and evaluation*, A K Peters, Natick, MA, 1998.

[4] T. May and M. Woods, "Alpha-particle-induced soft errors in dynamic memories," *IEE Trans. Electron Devices*, ED-26(1):2-9, 1979.

[5] J. F. Ziegler et al., "IBM experiments in soft fails in computer electronics," *IBM J. Res. Develop.*, 40(1):3-18, 1996.

[6] R. Gonzalez, B. M. Gordon, and M. A. Horowitz, "Supply and threshold voltage scaling for low power CMOS," *IEEE J. Solid-State Circuits*, 32(8):1210-1216, Aug. 1997.

[7] S. Borkar, "Design challenges of technology scaling," *IEEE Micro*, 19(4):23-29, July-Aug. 1999.

[8] M. Hamada and E. Fujiwara, "A class of error control codes for byte organized memory systems-SbEC-(Sb+S)ED codes," *IEEE Trans. on Computers*, 46(1):105-109, Jan. 1997.

[9] N. R. Saxena et al., "Fault-tolerant features in the HaL memory management unit," *IEEE Trans. on Computers*, 44(2);170-180, Feb. 1995.

[10] A. K. Somani and S. Kim, "Transient fault detection in cache memories by employing a small shadow cache," *Proc. of Dependable Computing for Critical Applications 6*, pp 19-39, 1998.

[11] R. E. Kessler, "The alpha 21264 microprocessor," *IEEE Micro*, 19(2):24-36, March-April 1999.

[12] N. Quach, "High availability and reliability in the itanium processor," *IEEE Micro*, 20(5):61-69, Sept.-Oct. 2000.

[13] A. Maamar and G. Russell, "A 32 bit RISC processor with concurrent error detection," *Proc. Euromicro Conf.*, pp. 461-467, 1998.

[14] J. Ohlsson and M. Rimen, "Implicit signature checking," *Int'l Symp. Fault-Tolerant Computing*, pp. 218-227, 1995.

[15] K. D. Wilkden ,"Optimal signature placement for processor-error detection using signature monitoring," *Int'l Symp. Fault-Tolerant Computing*, pp. 326-333, 1991.

[16] G. Miremadi and J. Torin, "Effects of physical injection of transient faults on control flow and evaluation of some software-implemented error detection techniques," *Dependable Computing for Critical Applications 4*, pp. 435-457, 1995.

[17] A. Mahmood, E. McCluskey, and Lu-D, "Concurrent fault detection using a watchdog processor and assertions," *Proc. Int'l Test Conf.*, pp. 622-628, Oct. 1983.

[18] I. Majzik, W. Hohl, A. Pataricza, and V. Sieh, "Multiprocessor checking using watchdog processors," *Computer Systems Science and Engineering*, vol. 11, no. 5, pp. 301-310, Sept. 1996.

[19] S. S. Yau and F. C. Chen, "An approach to concurrent control flow checking," *IEEE Trans. Software Engineering*, vol. SE-5, no.2, pp. 126-137, 1980.

[20] Z. Alkhalifa and V. S. Nair, "Design of a portable control-flow checking techniques," *Proc. High-Assurance Engineering Workshop*, pp. 120-123, 1997.

[21] M. A. Schuette and J. P. Shen, "Exploiting instruction-level parallelism for integrated control-flow monitoring," *IEEE Trans. Computers*, vol. 43, no. 2, pp. 129-140, Feb. 1994.

[22] J. H. Patel and L. Y. Fung, "Concurrent error detection in ALU's by recomputing with shifted operands," *IEEE Trans. Computers*, C-32, April 1983.

[23] G. Sohi and et. al. "A study of time-redundant fault tolerance techniques for high-performance pipelined computers," *Int'l Symp. Fault-Tolerant Computing*, 1989.

[24] J. Li and E. E. Swartzlander, "Concurrent error detection in ALUs by recomputing with rotated operands," *Proc. Int'l Workshop Defect and Fault Tolerance in VLSI Systems*, pp. 109-116, 1992.

[25] A. Mendelson and N. Suri, "Designing high-performance and reliable superscalar architectures-the out of order reliable superscalar (O3RS) approach," *Proc. Int'l Conf. Dependable Systems and Networks*, pp. 473-481, 2000.

[26] T. Sato and I. Arita, "In search of efficient reliable processor design," to appear in *Int'l Conf. Parallel Processing*, 2001.

[27] T. Sato and I. Arita, "Evaluating low-cost fault tolerant mechanism for microprocessors on multimedia applications," to appear in *Proc. Int'l Conf. Pacific Rim Dependable Computing*, 2001.

[28] T. M. Austin, "DIVA: a reliable substrate for deep submicron microarchitecture design," *Proc. Int'l Symp. Microarchitecture*, pp. 196-207, 1999.

[29] S. Chatterjee et al., "Efficient checker processor design," *Proc. Int'l Symp. Microarchitecture*, pp. 87-97, 2000.

[30] C. Weaver and T. Austin, "A fault tolerant approach to microprocessor design" *Proc. Int'l Conf. Dependable Systems and networks*, pp. 411-420, 2001.

[31] D. M. Tullsen, S. J. Eggers, and H. M. Levy, "Simultaneous multithreading: Maximizing on-chip parallelism," *Proc. Int'l Symp. Computer Architecture*, May 1996.

[32] E. Rotenberg, "AR-SMT: A microarchitectural approach to fault tolerance in microprocessor," *Proc. Int'l Symp. Fault-Tolerant Computing*, 1999.

[33] S. K. Reinhardt and S. S. Mukherjee, "Transient fault detection via simultaneous multithreading," *Proc. Int'l Symp. Computer Architecture*, pp. 25-36, 2000.

[34] F. Rashid, K. K. Saluja, and P. Ramanathan, "Fault tolerant though re-execution in multiscalar architecture," *Proc. Int'l Conf. Dependable Systems and Networks*, pp. 482-491, 2000.

[35] J. B. Nickel and A. K. Somani, "REESE: A method of soft error detection in microprocessors," to appear in *Proc. Int'l Conf. Dependable Systems and networks*, 2001.

[36] J. Clark and D. Pradhan, "Fault injection: a method for validating computer-system dependability," *IEEE Computer*, 28(6):47-56, June 1995.

[37] M. Hsueh, T. Tsai, and R. Iyer, "Fault injection techniques and tools," *IEEE Computer*, 30(4):75-82, April 1997.

[38] S. Chau, "Fault injection boundary scan design for verification of fault tolerant systems, " *Proc. Int'l Test Conference*, pp. 667-682, 1994.

[39] C. Constantinescu, "Teraflops supercomputer: architecture and validation of the fault tolerance mechanisms," *IEEE Trans. Computers*, 49(9):886-894, Sept. 2000.

[40] G. Miremadi and J. Torin, "Evaluating processor-behavior and three error-detection mechanisms using physical fault-injection," *IEEE Trans. Reliability*, 44(3):441-454, Sept. 1995.

[41] U. Gunneflo, J. Karlsson, and J. Torin, "Evaluation of error detection schemes using fault injection by heavy-ion radiation," *Proc. Symp. Fault-Tolerant Computing*, pp. 340-347, 1989.

[42] G. Miremadi et al., "Two software techniques for on-line error detection," *Proc. Symp. Fault-Tolerant Computing*, pp. 328-335, 1992.

[43] W. Moreno et al, "First test results of system level fault tolerant design validation through laser fault injection," *Proc. Conf. Computer Design: VLSI in Computers and Processors*, pp. 544-548, 1997.

[44] J. Samson et al, "A technique for automated validation of fault tolerant designs using laser fault injection (LFI)," *Proc. Symp. Fault-Tolerant Computing*, pp. 162-167, 1998.

[45] G. Choi, R. Iyer, and V. Carreno, "Simulated fault injection: a methodology to evaluate fault tolerant microprocessor architectures," *IEEE Trans. Reliability*, 39(4):486-491, Oct. 1990.

[46] G. Kanawati et al., "FERRARI: a flexible software-based fault and error injection system," *IEEE Trans. Computers*, 44(2):248-260, Feb. 1995.

[47] M. Rebaudengo et al., "Evaluating the effectiveness of a software fault-tolerance technique on RISC- and CISC-based architectures," *Proc. On-Line Testing Workshop*, pp.. 17-21, 2000.

[48] V. Sieh et al., "VERIFY: evaluation of reliability using VHDL-models with embedded fault descriptions," *Proc. Fault-Tolerant Computing*, pp. 32-36, 1997.

[49] J. Aidemark et al., "GOOFI: generic object-oriented fault injection tool," *Proc. Conf. Dependable Systems and Networks*, pp. 83-88, 2001.

[50] E. W. Czeck and D. P. Siewiorek, "Observations on the effects of fault manifestation as a function of workload," *IEEE Trans. Computers*, 41(5):559-566, 1992.

[51] S. Kim and A. K. Somani, "On-line integrity monitoring of microprocessor control logic," *Proc. Int'l Conf. Computer Design*, pp. 314-319, 2001.

[52] P. Duba and R. Iyer, "Transient fault behavior in a microprocessor-a case study," *Proc. Conf. Computer Design: VLSI in Computers and Processors*, pp. 272-276, 1988.

[53] E. W. Czeck and D. P. Siewiorek, "Effects of transient gate-level faults on program behavior," *Proc. Symp. Fault-Tolerant Computing*, pp. 236-243, 1990.

[54] M. Rimen and J. Ohlsson, "A study of the error behavior of a 32-bit RISC subjected to simulated transient fault injection," *Proc. Int'l Test Conf.*, pp. 696-704, 1992.

[55] G. Choi at el., "Fault behavior dictionary for simulation of device-level transients," *Proc. Conf. Computer-Aided Design*, pp. 6-9, 1993.

[56] N. Kanawati et al., "Dependability evaluation using hybrid fault/error injection," *Proc. Symp. Computer Performance and Dependability*, pp. 224-233, 1995.

[57] J. Guthoff and V. Sieh, "Combining software-implemented and simulation-based fault injection into a single fault injection method," *Proc. Symp. Fault-Tolerant Computing*, pp. 196-206, 1995.

[58] A. Amendola et al., "Fault behavior observation of a microprocessor system through a VHDL simulation-based fault injection experiment," *Proc. Conf. EURO Design Automation*, pp. 536-541, 1996.

[59] D. Gil et al., "Fault injection into VHDL models: analysis of the error syndrome of a microcomputer system," *Proc. Symp. Fault-Tolerant Computing*, pp. 418-424, 1998.

[60] J. Gaisler, "Evaluation of a 32-bit microprocessor wit built-in concurrent error-detection," *Proc. Symp. Fault-Tolerant Computing*, pp. 42-46, 1997.

[61] C. Kouba and G. Choi, "The single event upset characteristics of the 486-DX4 microprocessor," *Proc. Radiation Effects Data Workshop*, pp. 48-52, 1997.

[62] M. Franz, "The Java Virtual Machine: a passing fad?," *IEEE Software*, 15(6):26-29, Nov.-Dec. 1998.

[63] H. McGhan and M. O'Connor, "PicoJava: a direct execution engine for Java bytecode," *IEEE Computer*, 32(10):22-30, Oct. 1998.

[64] "The sun community source licensing program," Sun Microsystems, http://www.sun.com/micro-electronics/communitysource/.

[65] "PicoJava-II^{TM} microarchitecture guide," Sun Microsystems, http://www.sun.com/microelectron-ics/communitysource/picojava/.

[66] S. Dey et al., "Using a soft core in a SoC design: experiences with picoJava," *IEEE Design & Test of computers*, 17(3):60-71, July-Sept. 2000.

[67] "The java grande forum benchmark suite," EPCC, http://www.epcc.ed.ac.uk/javagrande/javag.html.

[68] P. Chow. *The MIPS-X RISC Microprocessor.* Kluwer, Boston, 1989.

[69] B. Cmelik and D. Keppel. Shade: a fast instruction-set simulator for execution profiling. *Performance Evaluation Review*, 22:128–137, May 1994.

[70] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, San Francisco, CA, 1996.

[71] H. Imai. *Essentials of Error-Control Coding Techniques.* Academic Press, San Diego, CA, 1990.

[72] J. Karlsson, P. Ledan, P. Dahlgren, and R. Johansson. Using heavy-ion radiation to validate fault handling mechanisms. *IEEE Micro*, 14(1):8–23, February 1994.

[73] J. M. Mulder, N. T. Quach, and M. J. Flynn. An area model for on-chip memories and its application. *IEEE J. Solid-State Circuits*, 26:98–106, February 1991.

[74] S. Park and B. Bose. Burst asymmetric/unidirectional error correcting/detecting codes. *Proc. Int'l Symp. Fault-Tolerant Computing*, pages 273–280, June 1990.

[75] D. Patterson, T. Anderson, N. Cardwell, R. Formm, K. Keeton, K. Kozyrakis, R. Thomas, and K. Yelick. Intelligent RAM (IRAM): Chips that remember and compute. *Proc. Int'l Symp. Solid-State Circuits*, pages 224–225, February 1997.

[76] J. C. Pickel and J. T. B. Jr. Cosmic ray induced error in MOS memory cells. *IEEE Trans. Nuclear Science*, NS-25:1166–1171, December 1978.

[77] A. M. Saleh. Reliability of scrubbing recovery-techniques for memory systems. *IEEE Trans. Reliability*, 30(1):114–122, April 1990.

[78] D. P. Siewiorek and R. S. Swarz. *Reliable Computer Systems: Design and Evaluation.* Digital Press, Beford, MA, 1992.

[79] A. K. Somani and K. Trivedi. A cache error propagation model. *Proc. Int'l Symp. Pacific Rim Fault Tolerant Computing*, pages 15–21, December 1997.

[80] P. Sweazey. SRAM organization, control, and speed, and their effect on cache memory design. *Midcon/87*, pages 434–437, September 1987.

[81] URL:. `http://www.specbench.org`.

[82] J. H. Edmondson et al., "Internal organization of the Alpha 21164, a 300-MHz 64-bit quad-issue CMOS RISC microprocessor," *Digital Tech. Journal*, vol. 7, pp. 119-135, 1995.

[83] G. Goldman and P. Tirumalai, "Ultra SPARC-II: the advancement of ultarcomputing," *Digest of Papers: Proc. Int'l Symp. COMPCON '96 Technologies for the Information Superhighways*, pp. 417-423, Feb. 1996.

[84] J. W. Bishop et al., "Power PC A10 64 bit RISC microprocessor," *IBM Journal of Research and Development*, vol. 40, no. 4, pp. 495-505, July 1996.

[85] D. Patterson, T. Anderson, N. Cardwell, R. Formm, K. Keeton, K. Kozyrakis, R. Thomas, and K. Yelick, "Intelligent RAM (IRAM): Chips that remember and compute," *Digest of Papers: Proc. Int'l Symp. Solid-State Circuits*, pp. 224-225, Feb. 1997.

[86] X. Castillo, S. R. McConnel, and D. P. Siewiorek, "Derivation and calibration of a transient error reliability model," *IEEE Trans. Computers*, vol. 31, pp. 6588-671, July 1982.

[87] S. J. Adams, "Hardware assisted recovery from transient errors in redundant processing systems," *Proc. 19th Ann. Int' Symp. Fault-Tolerant Computing*, pp. 170-175, June 1989.

[88] J. Sosnowski, "Transient fault tolerance in digital systems," *IEEE Micro*, vol. 14, pp. 24-35, Feb. 1994.

[89] J. J. Karlsson, P. Ledan, P. Dahlgren, and R. Johansson, "Using heavy-ion radiation to validate fault handling mechanisms," *IEEE Micro*, vol. 14, pp. 8-23, Feb. 1994.

[90] S. H. Sonawala, "An echo-back bus protocol for fault tolerant computer systems," *M.S. thesis, University of Washington*, 1988.

[91] A. K. Somani and S. Kim, "Transient fault detection in cache memories by employing a small shadow cache," *Proc. 6th Int' Symp. Dependable Computing for Critical Applications*, pp. 17-38, Mar. 1997.

[92] K. Boland and A. Dollas, "Predicting and precluding problems with memory latency," *IEEE Micro*, vol. 14, no. 4, pp. 59-67, Aug. 1994.

[93] S. A. Przybylski, *Cache and memory hierarchy design: a performance-directed approach*, Morgan Kaufmann Publishers, Inc., 1990.

[94] N. P. Jouppi and S. J.E. Wilton, "Tradeoffs in two-level on-chip caching," *WRL research report*, Western Research Laboratory, Nov. 1993.

[95] J. Handy, *The Cache Memory Book*, Academic Press, Inc., 1993.

[96] J. Hennessy and D. Patterson, *Computer Architecture a Quantitative Approach*, Morgan Kaufmann Publishers, Inc., 1996.

[97] A. Srivastava and A. Eustace, "ATOM: A system for building customized program analysis tools," *WRL research report*, Western Research Laboratory, Mar. 1994.

[98] J. M. Mulder, N. T. Quach, and M. J. Flynn, "An area model for on-chip memories and its application," *Proc. 19th Ann. Int' Symp. Computer Architecture*, pp. 181-190, May 1992.

[99] M. A. Schuette, J. P. Shen, Exploiting instruction-level parallelism for integrated control-flow monitoring, IEEE Trans. Computers 43 (2) (1994) 129–140.

[100] K. D. Wilken, J. P. Shen, Embedded signature monitoring: analysis and technique, Proc. Int'l Test Conf. (1987) 324–333.

[101] J. L. Hennessy, D. A. Patterson, Computer Architecture : A quantitative approach, Morgan Kaufmann Publishers, San Francisco, CA, 1996.

[102] M. Nicolaidis, "Design for soft-error robustness to rescue deep submicron scaling," *Proc. IEEE Int'l Test Conf.*, pp. 1140, 1998.

[103] T. J. Slegel et al. "IBM's S/390 G5 microprocessor design," *Micro*, 19(2):12-13, 1999.

[104] A. Wood, "Data integrity concepts, features, and technology," White paper, Tandem Division, Compaq Computer Corporation.

[105] P. K. Lala, *Self-Checking and Fault-Tolerant Digital Design*, Morgan Kaufmann, San Francisco, CA, 2001.

[106] E. Rotenberg. "AR-SMT: A microarchitectural approach to fault tolerance in microprocessors," *Proc. Int'l Symp. Fault Tolerant Computing*, pp. 84-91, 1999.

[107] S. K. Reinhardt and S. S. Mukherjee, "Transient fault detection via simultaneous multithreading," *Proc. Int'l Symp. Computer Architecture*, pp. 25-36, 2000.

[108] A. K. Somani and J. B. Nickel, "REESE: A method of soft error detection in microprocessors," *Prco. Int'l Conf. Dependable Systems and Networks*, pp. 401-410, 2001.

[109] D. C. Burger and T. M. Austin, "The simplescalar tool set, version 2.0," Technical Report CS-TR-97-1342, University of Wisconsin, Madison, 1997.

[110] URL: `www.simplescalar.org`.

[111] T. Williams et al., "SPARC64: A 64-b 64-active instruction out-of-order execution MCM processor," *IEEE J. Solid-State Circuits*, 30(11):1215-1226, 1995.

[112] N. Vasseghi et al., "200-MHz superscalar RISC microprocessor," *IEEE J. Solid-State Circuits*, 32(11):1675-1686, 1996.

[113] H. Sharangpani and H. Arora, "Itanium processor microarchitecture," *IEEE Micro* 20(5):24-43, 2000.