Demid Borodin

Performance-Oriented Fault Tolerance in Computing Systems

Performance-Oriented Fault Tolerance in Computing Systems

PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus prof.ir. K.C.A.M. Luyben,
voorzitter van het College voor Promoties,
in het openbaar te verdedigen

op donderdag 8 juli 2010 om 10:00 uur

door

Demid BORODIN

elektrotechnisch ingenieur
geboren te Samarkand, Sovjet Unie

Dit proefschrift is goedgekeurd door de promotor:
Prof. Dr. B.H.H. Juurlink

Copromotor:
Dr. S.D. Cotofana

Samenstelling promotiecommissie:

Rector Magnificus, voorzitter	Technische Universiteit Delft, NL
Prof. Dr. B.H.H. Juurlink, promotor	Technische Universität Berlin, DE
Dr. S.D. Cotofana, copromotor	Technische Universiteit Delft, NL
Prof. Dr. A.J.C. van Gemund	Technische Universiteit Delft, NL
Prof. Dr. G.J.M. Smit	Universiteit Twente, NL
Prof. Dr. C.I.M. Beenakker	Technische Universiteit Delft, NL
Dr. Z. Kotásek	Brno University of Technology, CZ
Prof. Dr. K.G. Langendoen, reservelid	Technische Universiteit Delft, NL

CIP-DATA KONINKLIJKE BIBLIOTHEEK, DEN HAAG

Demid Borodin

Performance-Oriented Fault Tolerance in Computing Systems
Delft: TU Delft, Faculty of Elektrotechniek, Wiskunde en Informatica - III
Thesis Technische Universiteit Delft. – With ref. –

Met samenvatting in het Nederlands.

ISBN 978-90-72298-07-2

Subject headings: fault tolerance, performance, reliability, fault detection, error detection.

Printed in The Netherlands

To my teachers, colleagues, friends, and family

Performance-Oriented Fault Tolerance in Computing Systems

Demid Borodin

Abstract

In this dissertation we address the overhead reduction of fault tolerance (FT) techniques. Due to technology trends such as decreasing feature sizes and lowering voltage levels, FT is becoming increasingly important in modern computing systems. FT techniques are based on some form of redundancy. It can be space redundancy (additional hardware), time redundancy (multiple executions), and/or information redundancy (additional verification information). This redundancy significantly increases the system cost and/or degrades its performance, which is not acceptable in many cases.

This dissertation proposes several methods to reduce the overhead of FT techniques. In most cases the overhead due to time redundancy is targeted, although some techniques can also be used to reduce the overhead of other forms of redundancy. Many time-redundant FT techniques are based on executing instructions multiple times. Redundant instruction copies are created either in hardware or software, and their results are compared to detect possible faults. This dissertation conjectures that different instructions need varying protection levels for the reliable application execution. Possible ways to assign proper protection levels to different instructions are investigated, such as the novel concept of instruction vulnerability factor. By protecting critical instructions better than others, significant performance improvements, power savings, and/or system cost reductions can be achieved. In addition it is proposed to employ instruction reuse techniques such as precomputation and memoization to reduce the number of instructions to be re-executed for fault detection.

Multicore systems have recently gained significant attention due to the popular conviction that the instruction level parallelism has reached its limits, and due to power density constraints. In a cache coherent multicore system the correct functionality of the cache coherence protocol is essential for the system operation. This dissertation also proposes a cache coherence verification technique which detects faults at a lower cost than previously proposed methods.

Acknowledgments

The story began when I was lucky enough to start my Master of Science thesis project with Dr. Ben Juurlink. He recognized some potential in the big lazy guy, and together with Professor Stamatis Vassiliadis[†] arranged the PhD study. Due to some timing and bureaucratic issues this was far not easy to organize, and I am very thankful to both Ben and Stamatis for their effort. Furthermore, Ben provided me with a great support throughout the whole PhD study. I greatly enjoyed the working experience with Ben, who made many significant contributions to my work, to my knowledge and skills. All the research-related skills I possess now have been learned from Ben: academic thinking, writing etc., and even academic traveling by train with a folding bicycle.

I would also like to thank all the members of my thesis committee for their interest in my work. Moreover, for their desire to participate in the committee, and for the useful remarks improving the thesis.

Furthermore, I would like to thank all my colleagues and professors in Computer Engineering group. With them I could enjoy a nice friendly atmosphere while working in our group. I also learned many useful things and skills, both related to work and others, from them. I am especially grateful to Lidwina Tromp for her invaluable help with the complex administrative issues that quickly turn me into the depressed mode. Thanks to the quick help from our system administrators Bert Meijs, Erik de Vries, and Eef Hartman, I never wasted time because of technical problems. Thanks to my roommate Asadollah Shahbahrami who shared many top secrets with me, I could easily adapt to the PhD life. Thanks to my friends, among others Christos Strydis, Carlo Galuzzi, and Eduard Gabdulkhakov, I always knew whom to contact when I needed some support and social life.

I am very grateful to my girlfriend Olga Kleptsova for tolerating and supporting me during all this time. She shared all my passions, such as photography and wind surfing, and even allowed me to turn the leaving room in our house into a small photo studio. Far not every girl would sacrifice so much for a crazy guy. I also thank my parents, my brother, and the whole our family for

their support and understanding.

Finally, I would like to express my gratitude to the SARC project of which I was a part during the PhD study, and to the Delft University of Technology, who accepted me as an MSc and later as a PhD student.

Demid Borodin Delft, The Netherlands, April 2010

Table of Contents

List of Tables

List of Figures

List of Acronyms and Symbols

ALU	Arithmetic Logic Unit
CMOS	Complementary Metal-Oxide-Semiconductor
CMP	Chip Multiprocessor
CPI	Cycles Per Instruction
CPU	Central Processing Unit
DIVA	Dynamic Implementation Verification Architecture
ECC	Error Correcting Codes
FT	Fault Tolerance
FU	Functional Unit
ILCOFT	Instruction-Level Configurability of Fault Tolerance
ILP	Instruction-Level Parallelism
I/O	Input/Output
IPC	Instructions Per Cycle
IVF	Instruction Vulnerability Factor
MTBF	Mean Time Between Failures
PC	Program Counter
POFT	Performance-Oriented Fault Tolerance
SEC-DED	Single Error Correcting and Double Error Detecting
SIFT	Software Implemented Fault Tolerance
SMT	Simultaneous Multithreading
TMR	Triple Modular Redundancy
VLIW	Very Long Instruction Word
VLSI	Very Large-Scale Integration

1 Introduction

Experience shows that in spite of all efforts to avoid it, manufactured computer systems tend to produce errors [1]. Fault avoidance [1] (also called fault-intolerance [2]) techniques, such as a thorough design process, using highly reliable materials, increasing the frequency and voltage margins, and post-manufacturing testing, are not able to prevent or catch all faults, or incur significant costs.

As the complexity of systems grows, the number of unfixed development faults increases. Because of the high design complexity, it is practically impossible to find all specification mistakes; because of the high complexity of produced systems, even very sophisticated testing methods fail to reveal all implementation mistakes. These are called *specification* and *implementation* mistakes [3] or *development faults* [1]. Malicious and non-malicious faults [1] can be made by designers, as well as by human operators working with a system at the operation stage.

In addition to the development faults, there are faults that occur during the system lifetime. These are physical faults, which can be internal and external (interaction) faults [1]. Examples of physical faults that can occur during a system's operation are noise-related faults [4], induced by electrical disturbances between the wires, and the faults triggered by environmental extremes such as fluctuations in temperature and power supply voltage, and radiation interference (gamma rays arriving from space [5] and alpha particles [6]).

The successful construction of dependable systems (systems able to deliver service that can justifiably be trusted [1]) requires applying both fault avoidance and fault tolerance (FT) techniques. Avizienis [7] argues that applying either of them alone leads to inefficiency and limited dependability. For example, applying fault avoidance by means of using highly reliable components is very expensive by itself. If these components still fail and there is no FT

mechanism which is able to keep the system operating correctly for a certain time after a failure, immediate manual maintenance is required to recover the system. Hence, maintenance personnel must always be present near a critical system, which adds a lot of expenses. At some point investing in a system's FT at the design stage becomes more cost-effective and efficient than increasing the investments in fault avoidance [8].

After the switch from vacuum tubes to the significantly more reliable transistors and until recently, strong FT features could typically be found only in special-purpose expensive computing systems that required extremely high reliability, such as computers used for military purposes and controlling aircrafts. The technology reliability was considered sufficient for commodity systems, and only a few FT techniques, such as Error Correcting Codes (ECC) [9] in memory, were usually used. However, modern technology trends such as shrinking the feature size and increasing the integration level have increased the probability of faults [4]. As a result, FT features are becoming necessary even in personal computers and embedded systems.

Avizienis [10] defines FT as "the ability to execute specified algorithms correctly regardless of hardware failures and program errors". There are several works introducing FT, defining its basic concepts and presenting a taxonomy [1, 2, 7, 8, 10–18].

This dissertation investigates low-cost FT techniques suitable for the design of systems in which expensive protection techniques are not affordable, such as general purpose personal computers and embedded systems. For these systems, it is not acceptable, for instance, to double the cost or to halve the performance in order to achieve reliability. Several techniques proposed in this work are able to reduce the FT integration cost for such systems. Most of the proposed methods reduce the performance degradation of existing time-redundant FT techniques. Because they target performance improvement, they are called *Performance-Oriented Fault Tolerance* (*POFT*) techniques in this thesis.

1.1 Problem Statement and Objectives

FT is achieved by introducing redundancy. Redundancy can appear in different forms. It can be space redundancy (additional hardware), information redundancy (additional information helping to verify some data), and time redundancy (multiple sequential executions of the same code, and/or execution

of additional verification code). Combinations of these redundancy types are also possible.

High-end critical computing systems, such as those used in aviation, nuclear plants, and for military purposes, typically use very expensive massive space redundancy. Multiple identical components, such as whole processors or their parts, are used to verify each other, detecting and possibly tolerating faults. Such a method provides a performance similar to that of a non-redundant system, but at a very high cost. The cost of the system, as well as its power and energy consumption, easily exceeds the cost of the non-redundant system by multiple times. While such a high cost is justifiable for critical-mission systems, this type of redundancy is in most cases not suitable for low-cost systems. Information redundancy (parity codes, ECC, etc.) is used both in expensive and relatively low-cost systems. It is mostly suitable for memory structures, however, and is usually unable to sufficiently protect the whole system. Time redundancy is often used in low-cost systems. It can significantly degrade the system performance, however, and can also increase the energy consumption. Hybrid FT techniques incorporating different forms of redundancy are very useful for both high-end and low-cost systems.

A careful design of FT techniques, employing (possibly combinations of) different redundancy types, is able to achieve significant reliability improvements at low cost and/or relatively little performance degradation. This is the main objective of this dissertation. Different approaches can be taken to achieve this goal. For example, unique system characteristics can be employed. Some systems are often unable to effectively utilize all their available resources. For example, the lack of Instruction-Level Parallelism (ILP) often leads to underutilized resources in superscalar processors. These idle resources can be used for FT purposes, thereby significantly reducing the performance penalty due to FT (see Section 2.4). A novel approach proposed in this dissertation (Chapter 3) questions what to protect, instead of how to protect it. It is shown that by protecting only the critical application parts, the overall application reliability can be achieved with a significantly reduced cost and/or performance degradation, or a higher reliability can be achieved at the same cost. Another approach employs performance improvement techniques, such as instruction precomputation and memoization, to reduce the FT overhead (see Chapter 5). Finally, low-cost function-specific protection methods can be designed, such as the cache coherence verification method proposed in Chapter 6.

1.2 Organization and Contributions

This dissertation makes several contributions to the FT overhead reduction.

Chapter 2 presents background information. First, it includes a brief history of FT development to introduce the reader to the subject. Then, existing POFT techniques for different architectures are presented. The ILCOFT approach proposed in this dissertation (Chapter 3) can be used with many of these techniques.

Chapter 3 introduces the Instruction-Level Configurability of Fault Tolerance (ILCOFT) approach. ILCOFT is based on the observation that different application parts differ in how critical they are for the overall application reliability. If more critical application parts are protected better than less critical parts, the same overall reliability can be achieved (or even improved) with a reduced FT overhead. To achieve this goal, ILCOFT requires a programmer to assign the proper protection to different application parts. Chapter 3 demonstrates when and why ILCOFT is useful, discusses how it can be implemented manually by a programmer, and automatically by a compiler. Then the ILCOFT concept is experimentally evaluated.

The greatest challenge ILCOFT faces is the difficulty to effectively assign proper protection levels to different application parts. The manual method requires significant programming effort and is very error-prone. The automatic compiler method is based on the assumption that only control flow instructions need to be protected, which is not safe for many applications. This problem is addressed in Chapter 4, where the novel Instruction Vulnerability Factor (IVF) concept is introduced. IVF measures how much of the final application output is corrupted due to fault(s) in every particular instruction. IVF is computed offline, once per application, and is then used by ILCOFT to determine the proper protection level for every executed instruction. As the experiments in Chapter 4 show, IVF provides a more accurate required protection level estimation than manual and compiler-based methods.

Chapter 5 proposes to employ the instruction precomputation [19] and memoization [20–22] techniques to reduce the fault detection overhead. Instead of duplicating every instruction for fault detection, this Chapter 5 proposes to compare the obtained instruction results with the corresponding precomputed and/or memoized results, when possible. Both instruction precomputation and memoization improve the performance and fault coverage compared to the duplication-based system. In addition, instruction precomputation is shown to improve the fault coverage, and in some cases the performance, of a previously

proposed memoization-based scheme. Moreover, a combination of precomputation and memoization is shown to outperform any one of these techniques used alone.

Chapter 6 addresses the multiprocessor-specific issue of cache coherence verification. For systems supporting cache coherence, the correct cache coherence operation is essential for the overall system reliability. Chapter 6 presents a low-cost verification scheme which outperforms previously proposed similar techniques at the expense of a slightly reduced fault coverage.

Finally, Chapter 7 summarizes the work presented in this dissertation and draws conclusions. In addition, possible future research directions are proposed.

2

An Overview of FT in Computing Systems

This chapter introduces the reader to FT in computing systems. First, a brief history of FT is provided in Section 2.1. Then, traditional FT techniques are presented in Section 2.3. Most existing FT techniques are based on the fundamental principles described in this section, such as duplication with comparison for fault detection, and N-modular redundancy for fault detection and recovery.

Thereafter, the chapter focuses on low-cost FT techniques suitable not only for expensive high-end systems, but also for low-cost general-purpose and embedded systems. Section 2.4 surveys existing POFT techniques. Many of these techniques can be enhanced with the ILCOFT approach presented in Chapter 3 to further reduce the FT overhead. SIFT techniques are presented in Section 2.5. SIFT methods are attractive because they do not require any hardware modifications, and thus, can be implemented faster and at a lower cost. Section 2.6 discusses FT techniques addressing the multiprocessor-specific cache coherence verification problem. Finally, Section 2.7 summarizes this chapter.

2.1 A Brief History of Fault Tolerance

Early computers, appearing from the 1940s until the late 1950s, were known to suffer from a lack of reliability [23, 24], since they were made from relatively unreliable vacuum tubes. For example, the first all-purpose digital electronic computer, ENIAC [25], which appeared in 1946, featured a Mean Time Between Failures (MTBF) of 7 minutes [26], and solved the given problems correctly only 54% of the time during a 4-year period [27]. Despite this, ENIAC was considered a pioneering device in reliability as it incorporated

18000 tubes, while at that time many experts found 300 tubes to be the maximum limit providing a feasible reliability [27]. Not surprisingly, FT gained significant attention at that time. The EDVAC computer, designed in 1949, is considered to be the first computer featuring duplicated ALUs whose results were compared to detect faults [24]. Another early computer, the Univac I [28], appeared in 1951. The Univac I extensively used ECC and parity checking for state elements, as well as duplicated arithmetic units and registers. The Univac I is considered to be the first computer using memory scrubbing: every 5 seconds all memory contents were verified by a parity checker.

The IBM systems featured several failure detection and recovery strategies [29, 30]. In the early systems (beginning 1950s), maintenance personal had to be available during the operation time. This ensured that if the system malfunctioned, an engineer with a deep knowledge of the system could recreate the failure and isolate the fault. Identifying the source of the problem involved running diagnostic programs, and if this did not help, an oscilloscope with logic diagrams was used for signal tracing. Fault isolation was usually performed by replacing a suspect unit with a spare one. Later (end of 1950s - beginning 1960s), as the system complexity grew, this method was found to be infeasible. It became too difficult to create diagnostic software with good coverage and precision (identifying the source of the problem sufficiently precise), and to trace errors in complicated designs. The concept of self-testing hardware was born. Following the terms in [30], the "failure capture" era came in place of the "failure recreation" era. When an error was detected, the operation terminated providing the information needed for error analysis. Finally, a transparent recovery without processing termination entered the scene, introducing the "failure recover" era. The IBM 650 system (mid-1950s, vacuum tubes) used error detecting codes, parity checks, duplicated certain circuitry, and utilized some software techniques such as checks for invalid operation codes and addresses, overflows, and others.

The first theoretical work on FT is attributed to John von Neumann [10] in his lectures given in 1952 and in [31], where the concept of *Triple Modular Redundancy* (*TMR*) was developed and analyzed. After a relatively slow progress in the late 1950s, two conferences on FT topics appeared in 1961 and 1962, which stimulated further developments in the field [10].

The second generation of computers, using semiconductor digital elements, appeared in the late 1950s [23]. Transistors feature a much higher reliability than the relays and tubes used before [24]. As the reliability of the components improved, the overall reliability of computing systems enjoyed a level

never seen before. This led to a temporal decrease of attention to computer FT (more precisely, to automated fault recovery) for non-critical systems. Fault detection/isolation techniques were still being used since they minimized the system downtime when a fault occurred [12]. The I/O equipment was protected by error detection and/or correction mechanisms [24]. In general, designers concentrated on the speed of computing systems, avoiding redundancy to keep the costs low, trusting the transistors to provide sufficient reliability [3].

In the late 1960s computers began to play key roles in space, military, transport, and other life-critical missions [32]. Because the potential cost and danger of malfunctions was extremely high, redundancy was employed again to implement automatic fault recovery [3, 12]. Since that time, the field of fault tolerant computing attracted a lot of research interest and was rapidly developing [3]. However, FT techniques have been applied mostly to special-purpose computers designed for critical missions.

The IBM 7030 (early 1960s, transistors, also called Stretch) used Single Error Correcting and Double Error Detecting (SEC-DED) codes in memory, parity checks and modulo-3 residue checks in arithmetic units and data paths. In addition, IBM 7030 duplicated some circuitry. A notable achievement in FT was the AN/FSQ-7 system (late 1950s, vacuum tubes), developed for air defense purposes, which had a spare processor running in parallel with the main one, executing test programs to test itself. Its memory was kept consistent with that of the main processor, so if the main processor detected a failure, the standby could continue operation. The I/O system also used standby spares. The IBM System/360 (mid-1960s) introduced a number of novel error detection techniques, such as two-rail checkers [33], scan-in and scan-out process, and microdiagnostics [34]. Several models of the System/370 (early 1970s) were the first to implement the CPU instruction retry. They also introduced autonomous diagnostic processors, which are also called watchdog processors. More information on IBM systems and their evolution can be found in [29, 30, 35, 36].

The third generation of computer systems (1969-1977) introduced integrated circuits. In the 1980s, the fourth generation appeared, when Very Large-Scale Integration (VLSI, integrated circuits with more than 30,000 transistors) became common [23]. The technology advances brought new trends [4]. To increase performance and decrease cost, designers attempted to reduce the feature size and power consumption. This inevitably increases the soft error rate of logic components [37]. From another point of view, the reduced cost of logic allows more protective redundancy, in some cases making the implementation of FT techniques very practical and cost effective [3]. VLSI features different

failure modes than previous technologies [3]. While small scale integration (integrated circuits with fewer than 30 transistors) and medium scale integration (30 to 300 transistors) mostly experienced faults on the pins and packaging, in large scale integration (300 to 30,000 transistors) and VLSI there are more internal faults. Assuming occurrence of only single faults is not sufficiently adequate in VLSI, because the smaller feature sizes lead to a larger amount of logic corrupted by external disturbances. Finally, the higher complexity leads to an increasing number of design mistakes. These new trends lead to the application of FT techniques not only in critical, but even in general purpose systems, and to an increasing research attention to FT [30].

We give a few examples of later FT systems, mostly targeted at critical missions. The Compaq NonStop Himalaya Servers [38], which appeared in early 1990s, have the *fail-fast* design philosophy: any faulty hardware or software module must detect the problem immediately and stop the operation to prevent fault propagation. The Compaq NonStop Himalaya uses lockstepping: two processors execute the same instructions and check each other every clock cycle. The memory is protected with SEC-DED codes, which is also able to detect three or four faulty bits in a row. In addition, a memory "sniffer" runs in the background every few hours, which tests the whole memory for latent faults. All data buses, I/O controllers, communication between processors and peripherals, and disk drives are protected by cyclic redundancy check, checksums, and other error detection techniques. The operating system and other system software is designed specifically for the system, thereby further enhancing the data integrity.

In 1994 IBM shifted from bipolar to CMOS technology in its mainframe systems. There had been several generations of CMOS System/390 (S/390) mainframes, implementing the ESA/390 (Enterprise Systems Architecture/390) instruction set architecture, until the end of the 1990s. The S/390 is a successor of the S/360 that first appeared in 1964. The first three generations of CMOS systems could not compete with the latest bipolar systems, neither in performance nor in FT. The IBM G4 [39, 40] (fourth generation S/390) system, announced in 1997, is the first which achieved a performance and FT level at least equivalent to those of the latest bipolar machines. Although CMOS technology features considerably larger fault avoidance characteristics than the bipolar technology, G4's design goals necessitated integrating significant FT features. Statistics show that the most powerful bipolar system, the 9X2 (9020 system generation), features a mean time between unplanned system downtime of 10 to 16 years. Bipolar IBM mainframe designs extensively use inline checking techniques [41] such as parity prediction. A parity predictor receives the same

inputs as the FU it checks (for example, the ALU) and calculates the expected parity of the output. This predicted parity is compared to the parity of the unit's output. The use of parity predictors, and inline checking in general, is not considered feasible for the G4, because it affects the cycle time and/or increases the chip area, wire length etc. [40]. Instead of inline checking, duplication of the entire I-unit (which fetches and decodes instructions) and E-unit (with execution elements) in a lock-step manner, with a comparison of all output signals every clock cycle, is preferred in the G4. In this way, neither the cycle time nor the CPI is affected, because the critical path is not lengthened, and the comparison of the results is overlapped in the instruction execution pipeline. The outputs of the duplicated I and E-units are compared at the *(N-1)*st stage of the instruction pipeline, and if they match, at the *N*th (final) pipeline stage the ECC-protected checkpoint array in the recovery unit *R-unit* is updated for future recovery needs. If an error (mismatch of outputs of the duplicated units) is detected, the final pipeline stage of the failed instruction is blocked, the CPU (except the R-unit) is reset, the parity-protected write-through on-chip L1 cache is flushed and loaded again using the contents of the ECC-protected off-chip L2 cache. In the mean time, the contents of the ECC-protected R-unit are checked, and if a permanent error is deduced, the CPU is stopped (the clock is stopped on the chip) since it is unable to recover. Otherwise, the CPU is set to the last checkpoint (state known to be good) and starts execution from the failed instruction. If the instruction fails again, the fault is considered to be permanent rather than transient (temporary), and the CPU is stopped. This recovery algorithm is invoked for any type of hardware error in the CPU. All the inner CPU recovery in the G4 is implemented in hardware, as opposed to microcode in the 9X2. In the case a failed CPU is stopped, the task which it ran is assigned to another CPU, with all the state information preserved. If a spare CPU is not available, graceful degradation of the system takes place. This process involves operating system intervention. The total CPU area overhead for FT in the G4 is estimated at 40%, while in the 9X2 the inline error detection caused an area overhead of about 30% [40]. Since the on-chip L1 cache is write-through, all the data it contains is also available in the L2 cache or main memory. Thus, parity protection is considered sufficient for the L1 cache. The L2 cache can hold the only valid copy of data, hence it needs higher protection and uses (72,64) SEC-DED codes. In the main memory, each chip provides 4 bits of a stored word, so an ECC able to correct a 4-bit error is used (a chip failure can be tolerated). The main memory uses background scrubbing (periodic checks) to make sure the data it stores is correct. Chips with detected permanent faults are marked as faulty and are not used after that. For the I/O

system, inline checking and some other techniques using space redundancy are utilized. The power and cooling systems also use space redundancy. At the end of 1997, the G4 experienced a mean time to unplanned system downtime of 22 to 26 years. More details on the FT features of the IBM S/390 G4 can be found in [40]. In the next generation S/390 system, the G5 [42], the FT further improved. The major difference with the G4 is that the G5 used a new mechanism of migrating tasks from a failed CPU to a spare. Unlike in the G4, this process does not involve operating system intervention, it is transparent to software. In the main memory system of the G5, unlike the G4, each chip provides only one bit of a word, so the lower cost (72,64) SEC-DED code is sufficient to correct a chip failure. Some FT enhancements in the I/O system are also introduced in the G5. A complete overview of the FT features of the G5 is presented in [43]. Bartlett and Spainhower compare the IBM zSeries and the Compaq NonStop Himalaya in [44].

Most superscalar processors are designed with the emphasis on high performance and lack FT features, trusting the reliable technology. The FT techniques are usually confined to the memory systems protected by ECC. There are a few exceptions, however. The HaL SPARC64 [45], which appeared in the mid-1990s, is a superscalar processor that features significant FT. The HaL-R1 computer system consists of a SPARC64 CPU and several other chips packaged on a silicon substrate multi-chip module. The HaL SPARC64 CPU utilizes a checkpointing mechanism similar to the one presented by Hwu and Patt [46] to restore the CPU state when errors and exceptions are detected while an instruction is executed. In case an error is detected, it is immediately reported to the precise state unit in the CPU, which stops issuing new instructions, kills all currently executing instructions, and brings the processor to the oldest checkpoint state. The processor enters the special Reset Error and Debug (RED) state, and the diagnostic processor is initiated, which runs recovery software. The RED state means that the processor runs in a restricted and degraded mode, without speculation. The execution is in-order and all caches are disabled. If a catastrophic error occurs in the RED state, the system halts, but the processor's internal state can be read by the diagnostic processor for analysis. Except the parity-protected level-0 instruction cache, most error detection mechanisms in the HaL-R1 computer are implemented outside the CPU, at the chip interface. No FT features are integrated into the CPU's functional units. The chip crossing signals arriving to the CPU from the instruction and data caches and the memory management unit, residing on separate chips, are protected by parity checks. Other chip crossing signals are protected with ECC. The HaL SPARC64 contains a programmable watchdog timeout mechanism,

which protects the CPU against a hang condition. If no instructions have been committed in a specified number of clock cycles, the system can optionally be brought to the RED state, and control is given to the special software running on the diagnostic processor, which takes control over the recovery actions. Different classes of memory cells on the cache chips are protected with different techniques, or have no protection at all. The large storage cells, that keep state information, are considered not vulnerable to alpha particles, so they are not protected. The cells keeping the least recently used information are also not protected, since their failure does not affect the functionality, but only corrupts the cache line replacement. The cache data array uses SEC-DED codes. The tag is protected by parity. If an error in a data store is detected, the CPU issues a retry command. If the error is not correctable, the system is put into the RED state. The memory system is protected with SEC-DED for transient failures, and fault-secure code, which uses only 8 check bits for 64 data bits. This code is derived from the SEC-DED-S4ED code [47]. This code prevents ECC miscorrection when an 8-bit DRAM chip fails. A detailed description of the FT in the memory management unit is given in [48].

2.2 Taxonomy of Faults

Figure 2.1 classifies existing faults according to Avizienis [1]. This dissertation focuses on (mostly operational) natural hardware faults, transient and in several cases permanent. In some cases other classes of faults (such as development and human-made faults) can also be addressed by the proposed FT methods, but only partially.

Natural hardware faults can be caused by both internal and external phenomena. Internal processes causing physical deterioration can result in internal faults, such as cross talk between wires, wire wearing-out etc. External processes originating outside the system and physically interfering with it, possibly penetrating into the system, can result in external faults. This can be, for example, radiation, external magnetic fields, thermal influence, etc. Internal as well as external sources can lead to both transient and permanent hardware faults in the system. More information on the taxonomy of faults can be found in [1].

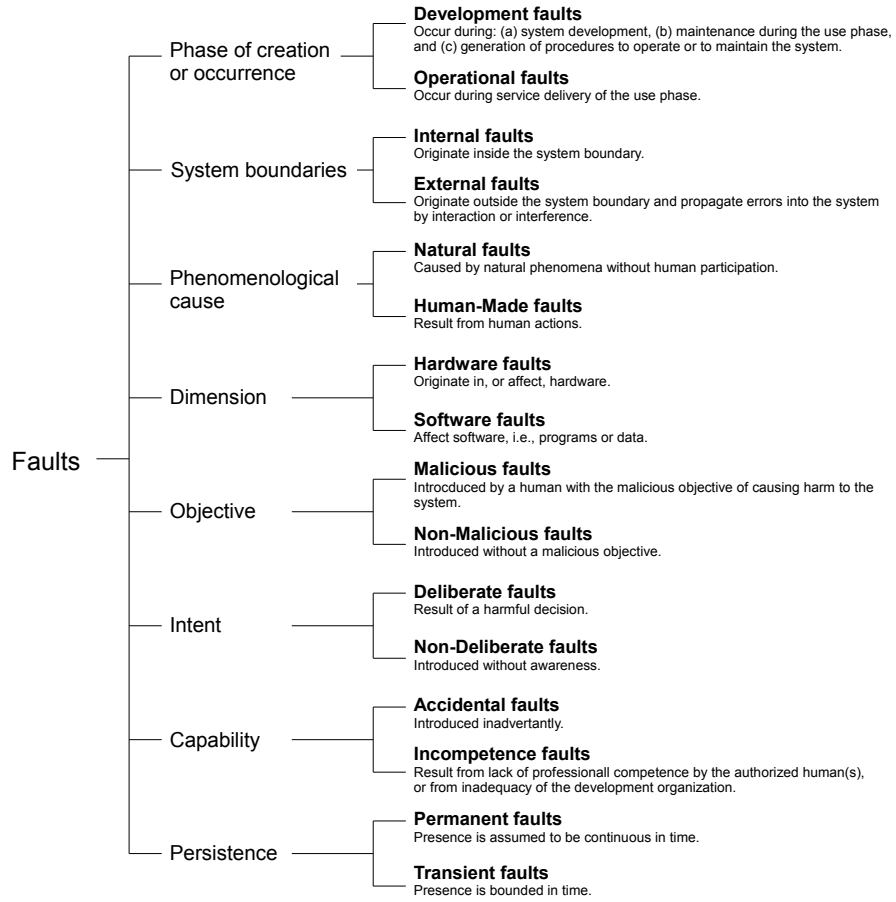


Figure 2.1: Faults classification according to Avizienis [1].

2.3 Conventional FT Techniques

FT is based on some form of redundancy. It can be in the form of *space*, *information*, or *time redundancy*. Space redundancy adds extra hardware resources to achieve FT. It usually leads to significant cost increases, but avoids performance degradation. Information redundancy adds some extra information, such as parity bits, for FT purposes. It needs additional resources and/or time to generate and use this information. Time redundancy performs an operation multiple times sequentially and compares the results. It does not increase the system cost, but significantly degrades performance. Any type of redundancy can appear in the form of additional hardware and/or software, which veri-

fies the functionality of the system. Different forms of redundancy are often combined to achieve the optimal result with the minimum overhead.

The cheapest form of FT is *active redundancy*, which is based on error detection followed by appropriate actions. For example, the faulty hardware unit can be disabled and its standby spare enabled, if it exists (otherwise, graceful degradation happens). The system will perform as long as at least one working unit is available. This approach typically does not correct the detected error, thus it is suitable only for systems that can tolerate a temporary erroneous result and is often used in long-life and high-availability systems. Error detection can be achieved by various techniques such as duplication with comparison, error detection codes, self-checking logic [33], watchdog timers, consistency and capability checks, and others [3, 35, 36].

The more expensive *passive redundancy* employs fault masking techniques, such as N-Modular Redundancy, ECC, masking logic, etc. Passive redundancy is more suitable for critical applications since it does not allow faults to propagate to the outputs.

The most expensive and the most effective is the *hybrid* form incorporating both passive and active redundancy. For example, the hybrid redundancy can use multiple identical hardware units verifying each other (providing fault masking), and spare unit(s). When a unit fails, it is replaced with a spare one, keeping the system protected. Figure 2.2 classifies the discussed types of redundancy.

The concept of *Triple Modular Redundancy* (*TMR*), one of the basic FT masking techniques, was developed and analyzed by John von Neumann in 1950s [31]. This scheme involves three identical blocks receiving the same inputs, which are expected to produce the same outputs. All the outputs are directed to a voter, which assumes that two or three matching values present the correct output, and masks the third one if it deviates, considering it to be faulty. TMR is able to detect multiple errors, as long as two outputs do not agree on a wrong value, and correct one error.

A reduced version of TMR is duplication with comparison, which is only able to detect errors, provided these errors do not affect the outputs of both blocks in the same way. TMR can be extended to *N-Modular Redundancy*, which uses N identical blocks, and performs a majority voting on the results. In TMR, N equals 3, and in duplication with comparison N equals 2.

The N-Modular Redundancy technique can be applied at any level, from discrete transistors to whole systems, as well as for any redundancy method (space, information, and time redundancy in hardware or software). The only

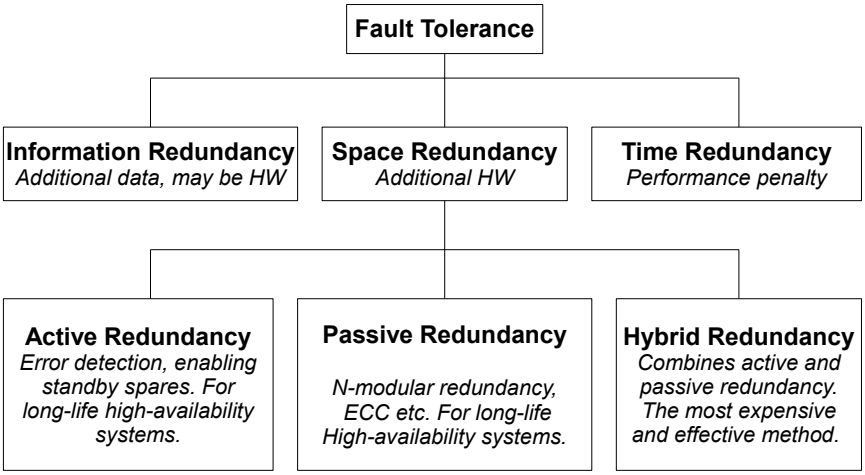


Figure 2.2: Protective Redundancy Types.

practical application of component-level redundancy (the discrete transistor level) is found in the PPDS computer used in NASA's Orbiting Astronomical Observatory (OAO-2) satellite launched in 1968 [49]. It is one of the latest computers assembled from discrete transistors [12]. PPDS utilizes masking (quadruple) redundancy at component level: instead of one transistor of a non-redundant system, there are four of them, implemented in such a way that a failed transistor is masked by the others. This technology is not adequate for integrated circuits because the independence of adjacent components' failures cannot be guaranteed. N-Modular Redundancy in software takes the form of *N-version programming* [50] (see Section 2.5 for details).

The weak spot of N-Modular Redundancy is the voter, which must provide a reliability level appropriate for the multiplicated module whose functionality it verifies. A weak voter makes N-Modular Redundancy useless because the output of multiple modules is not reliable. The reliability of voters has been studied [51] and some effort has been made to improve it. This is achieved, for example, by creating self-testing voters [52, 53] and by using a transistor redundancy approach [54], in which faults in the voter are masked at the transistor level. A voter which compares whole output words rather than separate bits has been proposed to minimize the risk of an improper agreement report [55]. In addition to the vulnerability of the voter, all pure N-Modular Redundancy techniques are susceptible to common failures [32]. Common

failures affect the outputs in the same way, so that all the modules produce the same erroneous output, which is accepted by the voter.

The basic form of time redundancy is recomputing (performing the same computation multiple times) with results comparison. This scheme aims at detecting transient (temporary) faults only. The problem of recomputing on hardware with a permanent fault is the same as that of simultaneous computing on multiple hardware units with common faults: the faults affect the results in the same way, the outputs match, so the error is not detected. However, there exist space redundancy schemes covering common faults, and time redundancy schemes covering permanent faults. These schemes change the form of the inputs (encode them) and expect to get matching results after performing an appropriate compensating transformation (decoding) of the output. These transformations guarantee that common and permanent faults affect results in different ways. Examples of such techniques are *alternating logic* [56], *alternate-data retry* [57], *recomputing with shifted operands* [58], *recomputing with rotated operands* [59], and *recomputing with swapped operands* [3]. There are also hybrid schemes combining hardware and time redundancy, such as *recomputing using duplication with comparison* [60] and its enhancements [61, 62].

In order to minimize the cost of the applied redundancy, Huang and Abraham [63] proposed algorithm-based FT, which utilizes the properties of particular algorithms. Algorithm-based FT designs provide a high level of FT at an extremely low cost compared to the universal methods discussed above. However, algorithm-based FT methods need to be designed specifically for every algorithm. Huang and Abraham [63] considered matrix operations as an example. Input matrices are encoded by adding a column and/or row containing the sum of all the elements in the corresponding row/column. Matrix operations are performed on these encoded matrices. The results are decoded, providing error detection and location.

For more information on traditional FT techniques we refer to [3, 35, 36].

2.4 POFT Techniques

In the late 1980s a new trend in the research on FT appeared. Researchers avoid introducing new hardware (space) redundancy. Instead, they introduce time redundancy and try to minimize the performance overhead by efficiently using the available but unutilized or underutilized hardware resources. We refer to these techniques as *Performance-Oriented Fault Tolerance* (*POFT*) techniques. This is possible with some modern architectures, such as superscalar,

Table 2.1: POFT Techniques for Different Architectures.

Target Architecture	POFT Techniques
VLIW	Using idle instruction slots. [64–67].
Superscalar	Using resources which are idle due to insufficient ILP. [67–78], [79, 80].
SMT	Duplicate thread verifies the results produced by the original thread. [81–86].

VLIW, and multicores, because most applications are not able to effectively utilize all available resources. For example, when running applications without sufficient thread-level parallelism, some cores in multiprocessor systems might not perform any useful work.

Table 2.1 characterizes the POFT techniques used for different machine organizations: VLIW, Superscalar, and SMT. Table 2.2 presents an overview of different POFT techniques, which are discussed in detail in subsequent sections.

Table 2.2: Overview of POFT Techniques.

Technique	Verification target	Method
[64], [65], [66].	VLIW datapath.	Duplicate VLIW instructions in software. [66] compares the results in hardware.
ARC [67].	Control flow in superscalar or VLIW.	Watchdog task checks control flow concurrently using signature in a special register.

Continued on Next Page...

Table 2.2 – Continued

Technique	**Verification target**	**Method**
[68].	FUs, possibly also other parts. Covers permanent faults.	Instruction duplication using RESO [58] for the duplicates.
[69].	FUs and possibly dynamic scheduler.	Instruction duplication.
[70].	Caches, fetch and decode units, dynamic scheduler, register file, result bus.	Various techniques.
[72].	Datapath in multiscalar architectures [87].	Task re-execution.
O3RS [73].	Different parts in superscalar architectures.	Instruction duplication. ECC protection of the instruction decoder, ROB, RAT, fetching and write-back mechanisms.
[74].	Superscalar datapath.	Execute instructions multiple times. Using mechanisms for speculative out-of-order execution to recover.
SHREC [75].	Superscalar datapath.	Redundant threads.
[76].	Superscalar datapath.	Instruction duplication. When possible, using instruction memoization instead of duplication.
[77].	Partial coverage in superscalar datapath.	Instruction duplication when idle slots are available, or memoization.

Continued on Next Page...

Table 2.2 – Continued

Technique	Verification target	Method
[78].	FUs, partial permanent fault coverage.	Instruction duplication. When possible, using instruction precomputation instead of duplication.
[79].	Datapath.	Using the properties of self-checking and semi self-checking instructions.
DIVA [71, 88–90].	Whole processor, also permanent faults.	Simple watchdog processor performing equivalent task.
[91], AR-SMT [81], SRT [82], SRTR [83], CRT [84], Slipstream processors [85, 86].	From FUs to the whole datapath.	SMT-based redundant threads.

2.4.1 FT in VLIW Architectures

VLIW architectures feature several operations in each instruction word. Only data-independent operations can appear in different slots of a single instruction. Furthermore, in many cases only a certain type of operations (utilizing a certain type of FU) is allowed to occupy each slot within an instruction word. These constraints often lead to a large number of unused slots in VLIW instructions, even when the application is reasonably optimized. The unutilized instruction slots can effectively be used for FT purposes, for example, by repeating operations and comparing the results. However, even if an operation is duplicated and scheduled in parallel with the original operation (in the same instruction but in different slots), the results should be compared and an error signal triggered in case of a mismatch. It is impossible to do so within the same instruction without changing the hardware. Thus, the comparison must appear later, occupying a certain amount of resources and likely decreasing performance.

Holm and Banerjee [64] are the first to propose to duplicate ALU instructions for error detection in a VLIW datapath. Bolchini [65] presents a similar dupli-

cation and comparison technique, which attempts to use only idle instruction slots for duplicates and checking operations, if possible. This reduces the performance penalty. Hu et al. [66] propose a compiler-directed instruction duplication scheme which duplicates instructions only if this does not exceed the maximum permitted performance degradation. To avoid additional comparison instructions, additional hardware support is used to compare the results of duplicated instructions.

Schuette and Shen [67] propose a technique called Available Resource-driven Control-flow (ARC) monitoring. Although the authors conduct experiments with a VLIW architecture, the technique is also applicable to superscalar systems. ARC creates a watchdog task which checks an application's control flow concurrently. The program is extended with instructions that regularly update a special register holding a "key" value. Other embedded instructions compare this value against the correct one generated by the compiler. Illegal execution sequences are detected when wrong key values appear. The watchdog task runs on the same processor and an effort is made to schedule its instructions to occupy only otherwise idle execution resources, in order to avoid introducing performance overhead. These idle resources can be both empty slots in VLIW as well as stalls in superscalar architectures. Several similar software-based techniques are presented in Section 2.5.

2.4.2 FT in Superscalar Architectures

In superscalar processors, FUs are typically deeply pipelined, and sometimes multiple instances of them are present. Some of these resources have a good chance of staying idle during stall cycles due to data dependencies between instructions.

Superscalar architectures have gained more research attention in the FT context than VLIW processors. One of the early works in this field, before the superscalar era, is performed by Sohi et al. [68]. For each instruction, a companion instruction is created using the Recomputing With Shifted Operands (RESO) technique [58]. Different strategies (places where the companion instructions can be generated, ways to compare results etc.) are analyzed. The strategy which is found to be the best for the considered single-issue scalar architecture extends the hardware of each FU to duplicate instructions, updates the register file as soon as (possibly incorrect) results are available to avoid stalling while waiting for a duplicate result, and degrades performance by only 0.21% on average.

One of the early works on incorporating FT features into superscalar processors is done by Franklin [69] in 1995. The author investigates ways to duplicate instructions at runtime and to compare the results to detect errors. Two places where instructions can be duplicated are presented and analyzed: (1) in the dynamic scheduler after an instruction is decoded, and (2) in an FU when it executes an instruction. Duplicating in the dynamic scheduler is preferred, since it incurs a smaller performance degradationand provides better fault coverage (the duplicate instructions can be executed on different FUs). The performance evaluation shows that with a superscalar issue width of four, the introduction of FT incurs a larger performance penalty than with an issue width of eight, because more instruction slots are wasted in an organization with a larger issue width.

In [70] Franklin continues his work on FT in superscalar processors, analyzing ways to deal with errors occurring in different parts of the processor: in the instruction and data caches, fetch and decode units, the dynamic scheduler, the register file, and the result bus. The FUs are not considered. Several FT techniques are proposed and evaluated from the fault coverage and performance points of view. The techniques do not require any changes to the instruction set architecture and compiler, and incur a low hardware and performance overhead. Among the proposed techniques is to keep the instructions' ECC bits in a separate *instruction check bit cache* rather than in the instruction cache itself. Two distinct address decoders are used to access a cached value and its ECC bits. At the commit stage, an instruction is re-encoded (performing the reverse of the instruction decoder operation), and verified using the check bits. In addition to detecting a value corruption in the instruction cache, fetch unit, and decoder, this technique is able to catch addressing errors, when the instruction cache returns an incorrect value (from a wrong address) to the fetch unit. The technique eliminates the need for a signature-based control flow checking which involves object code modification. Following the technique used in many IBM mainframes, Franklin proposes to use a *register check bit file* with ECC values protecting the contents of the register file. Similarly, the ECC check bits for the contents of the data cache are kept in *data check bit cache*. If an instruction's register operands are verified at the commit stage by recalculating the ECC bits and comparing them with the contents of the RCBF, several types of decoder, dynamic scheduler, and register file errors can be detected and/or corrected.

Rashid et al. [72] incorporate FT into multiscalar architectures [87]. Multiscalar architectures feature multiple parallel processing units that execute tasks derived from a single program. In [72], tasks are re-executed for error detec-

tion purpose. Static and dynamic task re-execution schemes are assessed. In the static scheme, a certain number of processing units are assigned a priory for the original task, and the others – for its duplicate task, whose results are compared to detect faults. In the dynamic scheme, the resources are distributed dynamically, giving priority to the trailing tasks when necessary. Simulation results demonstrate a performance degradation of 5%-15% due to the introduction of FT. The approach targets both transient and permanent faults.

Mendelson and Suri [73] introduce the Out-Of-Order Reliable Superscalar (O3RS) architectural approach, which uses time redundant FT technique for error detection and recovery. The approach mainly targets high performance with the minimum possible FT overhead. The authors suggest minimization of the FT enhancements in a superscalar processor, using traditional ECC when possible. It is argued that the lookup table-based instruction decoder, Re-Order Buffer (ROB), Register Alias Table (RAT), execution units, fetching and write-back mechanisms do not need any special FT enhancements except the ECC. Only the retirement mechanism needs to be extended to perform voting on the duplicated instructions' results. The ROB is modified in such a way that every instruction except loads and stores is executed twice and the results are compared. If the results of two instances of a duplicated instruction do not match, the instruction continues executing until the last two instances agree on the result (the complete results history is not kept to reduce hardware costs, so only the last two results can be compared). The approach actually addresses only transient errors in FUs.

Ray et al. [74] investigate a time-redundant FT technique for superscalar processors. The authors suggest that existing mechanisms for speculative out-of-order execution can be used for cheap transient fault detection and recovery. The register renaming hardware is adapted to issue multiple (two or more) instances of a single instruction, and their redundant results are compared in ROB before retiring to detect errors. The work extends the idea of duplicating instructions to issuing two or more instances of every instruction, and considers employing the voting mechanism on the differing outcomes when possible. If an error is detected, and the voting is not applied, the existing instruction-rewind mechanism is used for recovery, reverting to the last correct stage. The performance penalty of this recovery scheme is only on the order of tens of cycles. The performance impact of the proposed technique is evaluated both analytically and by simulation. The simulation results show that the instruction triplication scheme with majority voting on the results outperforms the instruction duplication scheme with the rewind-based recovery only for very high error rates. In addition, the simulation results demonstrate that in the ab-

sence of faults, integration of the proposed FT techniques causes a 2% to 45% reduction of the throughput.

Smolens et al. [75] analyze the performance bottlenecks of the redundant threads execution in superscalar pipelines. Based on the obtained results, they propose the SHREC (SHared REsource Checker) architecture which reduces the performance overhead by efficiently sharing resources. One of the key SHREC features is that the input operands are provided to the instructions in the redundant thread by the main thread. The redundant thread is managed by a separate in-order scheduler. This restores the full capacity of the issue queue and reorder buffer to the main thread, without physically enlarging them. In addition, the main and redundant threads can stagger elastically, up to the size of the reorder buffer. This reduces the resource contention between the threads, allowing the redundant thread to use the resources left idle by the main thread, rather than competing for the busy ones.

Parashar et al. [76] improve the performance of the instruction duplication approach [69, 74] by employing instruction reuse (also called *memoization*) [22, 92]. This technique avoids redundant computation, when possible, by reusing the previously saved results for the duplicated instructions. The original instructions are always executed normally, providing at the same time verification of the reuse buffer itself. Simulation results demonstrate an average reduction of 23% in the IPC loss compared to the full redundant computation (re-execution of all the instructions).

Gomaa and Vijaykumar [77] further reduce the performance degradation due to fault detection by issuing duplicated instructions only when idle computational resources are present in the system (low-ILP phases). They combine this with instruction reuse [76] when possible (when results are available in the reuse buffer). This way, a certain fault coverage is gained almost without any performance degradation.

In this dissertation (Chapter 5, also in [78]) the instruction precomputation technique [19] is used to improve the performance and fault coverage of instruction duplication, in a way somewhat similar to how memoization is used in [76]. Instruction precomputation is a work reuse technique involving offline application profiling. The profiling data (instruction opcodes with input operands and corresponding results) is stored together with the application binary code. Prior to execution, the profiling data is loaded into a special hardware buffer, and the stored results are used when possible, avoiding recomputation. The precomputation-based technique improves the long-lasting fault coverage of the memoization-based technique [76], and outperforms it for

some benchmarks. In Chapter 5 (and in [93]) instruction precomputation and memoization are combined to reduce the fault detection overhead, achieving better performance than any of these techniques achieves alone.

Kumar and Aggarwal [79] introduce *self-checking* and *semi self-checking* instructions. They observe that many instructions (38% on average for the simulated benchmarks) produce results that are identical to at least one of the input operands. An example of such an instruction is addition with zero as one of the operands. These instructions are called self-checking, because to verify the correctness of the output, it is sufficient to compare the input operand with the produced result. Semi self-checking instructions have at least one operand of a small size (only a few least significant bits are non-zero). In the proposed scheme, result verification for the self-checking instructions is performed after the first execution by comparing the output with the input operand. For the semi self-checking instructions, the upper bits of the result are compared against the input operand, and the lower bits are recomputed by a dedicated hardware. The other instructions are executed redundantly. The experimental results report the recovery of 51% of the performance and 40% of the energy consumption overhead due to redundancy.

This dissertation (Chapter 3, also in [80, 94]) presents an approach called Instruction-Level Configurability of Fault Tolerance (ILCOFT). It is based on the observation that different instructions often have different reliability requirements in a program. Consider, for example, a loop performing matrix addition. If the addition itself produces a wrong result, only one of the resulting matrix elements will be wrong. In large images, however, one wrong pixel value is tolerable. If, on the other hand, a fault affects the loop index calculation, a large portion of the resulting matrix will be corrupted, or the application will even crash. ILCOFT gives a programmer the opportunity to define the required FT degree for single or groups of instructions. ILCOFT-enabled version of EDDI [95] (see Section 2.5 for details), which duplicates only the critical instructions, is compared against the pure EDDI in the contents of several multimedia kernels and JPEG encoding application. Only one of the most computationally intensive kernels in the JPEG encoder is enhanced with ILCOFT. Both the performance and energy consumption are improved by up to 50% at the kernel level and 16% at the application level. This dissertation (Chapter 4) also introduces the notion of Instruction Vulnerability Factor (IVF). Using fault injection experiments, IVF evaluates how much of the final application output is corrupted due to faults in every instruction. This information is then used in ILCOFT-enabled system to assign appropriate protection level to every instruction.

2.4.3 Dynamic Implementation Verification Architecture

Austin [71] presents a novel microarchitecture-based FT technique called the Dynamic Implementation Verification Architecture (DIVA). The technique does not actually utilize the available redundancy, but applies the watchdog concept to superscalar processors. The DIVA consists of a speculative and a dependable part. The robust part, the *DIVA checker*, is meant to provide dependability of the whole system, checking and correcting the operation of the deeply speculative part, the *DIVA core*. The DIVA core is a high-performance microprocessor without the retirement stage, which delivers its speculative results (completed instructions in program order, along with their input operands) to the DIVA checker for verification. The DIVA checker re-executes the instructions, checking separately the FUs results and register and memory communication. If all checks succeed, the DIVA checker commits the results. If an error has occurred, the DIVA checker corrects the results, flushes the pipeline in the DIVA core, and resumes execution from the following instruction.

Because the DIVA core delivers the instructions together with their operands, the DIVA checker does not face dependency problems, hence its pipeline can be very simple and fast. Because of this, the system performance is not affected very much. The experimental results show that the average slowdown caused by the DIVA checker is only 3%. With some additional hardware enhancements, the slowdown even drops to 0.03%.

Austin argues that the simplicity of the DIVA checker and its fault avoidance techniques should make its design and production significantly cheaper than incorporating FT into a whole high-performance processor. At the same time, the presence of the dependable DIVA checker makes it possible to be less concerned about the dependability of the DIVA core, thereby decreasing its cost.

Subsequent work [88] analyzes the performance overhead caused by introducing the DIVA checker, and proposes several improvements to decrease it. Mneimneh et al. [89] propose a strategy for a formal verification of the DIVA checker. Verifying the correctness of the DIVA checker is sufficient for the whole DIVA system, since the checker corrects the errors occurring in the other part. Weaver and Austin [90] further analyze the performance impact of the DIVA checker, present its detailed design, and assess the area and power requirements. The results demonstrate virtually no performance penalties, and less than 6% area and 1.5% power overhead.

2.4.4 FT Based on Simultaneous Multithreading

Another proposed approach utilizes the Simultaneous Multithreading (SMT) [96] technique for FT purposes. Originally, SMT has been proposed to increase the utilization of the available resources in superscalar processors. When a single thread lacks sufficient ILP to utilize all resources, another thread can use them. An SMT processor executes multiple independent threads in the same clock cycle, using different FUs.

In a fault tolerant SMT processor, the original program is run as one thread, while its duplicate is executed as an independent redundant thread, and the results are compared. This scheme provides several advantages over conventional hardware replication (when two processors run the same program and compare results). First, it requires less hardware, because the processor does not have to be fully duplicated. Second, one thread can assist the other by prefetching data and providing branch prediction. Furthermore, as noticed in [91], the multithreaded approach can target both high-performance and high-reliability goals, if reconfiguration to either high-throughput or fault tolerant modes is allowed.

The potential of integrating low-cost FT into multithreaded processors by redundantly executing several identical threads is first recognized by Saxena and McCluskey [91]. They estimate the performance overhead at 20% for duplicated and 60% for triplicated thread.

Rotenberg [81] introduces the Active-stream/Redundant-stream Simultaneous Multithreading (AR-SMT) technique, which duplicates instructions when they reach the execution pipeline stage. The two instruction streams are processed separately in the SMT manner, and their results are compared. The technique mostly targets transient faults. Permanent faults are unlikely to be discovered as both instruction streams are executed on the same resources. The performance overhead of AR-SMT is evaluated for a trace processor [97] with four processing elements. A performance penalty of 12% to 29% is reported.

Reinhardt and Mukherjee [82] further investigate the FT capability of SMT processors, deriving a Simultaneous and Redundantly Threaded (SRT) processor. The work introduces the concept of *sphere of replication*, which abstracts both the physical and logical redundancy and helps to estimate the fault coverage of a system (determines what parts are protected and what parts are not). Two different spheres of replication for SRT processors are analyzed and some problems identified. Several hardware solutions addressing these problems and the performance overhead are proposed. The experimental re-

sults demonstrate that an SRT processor is able to deliver an average performance improvement of 16%, with a maximum of 29% compared to an equivalent hardware-replicated lockstepped solution, while providing a similar transient fault coverage. Vijaykumar et al. [83] extend the SRT with transient fault recovery capabilities and some hardware enhancements, introducing Simultaneously and Redundantly Threaded processors with Recovery (SRTR). Later, Mukherjee et al. [84] further investigate the SRT approach, applying it to single- and dual-processor simultaneous multithreaded single-chip devices. A more detailed processor model than in the previous work is used, revealing some issues in a single-processor device that were not addressed in [82]. The results show that the actual performance overhead caused by creating a redundant thread is on average 32% rather than 21%, as reported before. The paper extends the original SRT design with some performance optimizations, and a technique which increases the permanent fault coverage by forcing duplicated instructions to use space redundancy rather than time redundancy whenever possible (if multiple hardware resources exist). In addition, the work presents a new technique called Chip-level Redundant Threading (CRT), which applies the SRT approach to a CMP [98] environment, forcing the redundant streams to run on different cores. On single-thread workloads, when there is only one (leading) thread of each application, CRT is similar to lockstepping in that each core runs a single thread, and it achieves similar performance. When two redundant threads represent each of the two applications, every core runs the leading thread of one application and the trailing thread of the other. In this case, CRT outperforms the lockstepped solution by on average 13%, with a maximum of 22%.

2.4.5 Slipstream Processors

Sundaramoorthy et al. [85, 86] introduce slipstream processors, primarily targeting high performance, but also incorporating some FT capabilities. Slipstream processors utilize the unnecessary redundancy, repetition, and predictability that are observed in general purpose programs. The background work [99] defines *ineffectual* instructions, which can be removed from a program without affecting the final output. Examples of ineffectual instructions are those that write values to memory or registers that are never used later, write the same values which are already present at the destination, and highly predictable branches. These ineffectual instructions, and the whole computation chains leading to them (*ineffectual regions*), can be eliminated from a program, significantly reducing the amount of computations needed. In some

cases a reduced instruction stream can comprise only 20% of the original, equivalent instruction stream [99].

In slipstream processors, for each user program, two processes are created by the operating system. These redundant processes are executed simultaneously on a CMP or SMT system. The leading process (advanced stream, *A-stream*) runs slightly ahead of the trailing process (redundant stream, *R-stream*). Special hardware, the *instruction-removal detector*, monitors the R-stream and identifies the ineffectual instructions that might be eliminated in the future. The instruction-removal detector delivers this information to the *instruction-removal predictor*, which determines the reduced set of instructions to be executed in the A-stream at the future dynamic instances of these instructions. The A-stream performs fast because of the reduced number of instructions it has to process, and the trailing R-stream is sped up by the control and data flow outcomes provided by the A-stream. As a result, the two redundant streams run faster than each of them can do alone. The experimental results reported in [85] demonstrate an average performance improvement of 7% on a CMP. In [86], an improvement of 12% is achieved for a CMP, and 10% to 20% for an 8-way SMT.

Because of the speculative nature of the process which determines ineffectual instructions, necessary instructions can be skipped by the A-stream, corrupting its results. If the R-stream detects deviations in the A-stream, the architectural state of the R-stream is used for recovery of the A-stream. Thus, the R-stream plays the role of a watchdog processor. Faults occurring in the instructions executed redundantly in both the A-stream and the R-stream can be detected and corrected. However, not all the instructions appearing in the R-stream are also executed in the A-stream, so the fault coverage is partial. Flexibility is provided to the user or operating system, allowing to trade performance and fault coverage.

2.5 Software Fault Tolerance

Table 2.3 presents an overview of several software FT techniques. Software approaches to FT can be classified into those that target software (design) faults, and those that aim at detecting and/or tolerating hardware faults. Subsequent sections discuss them in detail.

Table 2.3: Overview of Software FT Techniques.

Technique	**Verification target**	**Method**	**Hardware support**
Recovery blocks [100], N-version programming [50].	Software design faults.	Multiple independent but functionally equivalent implementations.	No
CATCH [101], [102].	Depends on fault detection technique used.	Checkpointing performed by compiler.	No
SIS [103], CCA [104], ECCA [105], CFCSS [106], ACFC [107], YACCA [108].	Control flow faults.	Control flow signatures.	No
BSSC and ECI [109].	Control flow faults.	Control flow signatures, signaling instructions in invalid memory locations.	Possibly a watchdog timer.
[110], [111], SIC [112], [113].	Control flow faults.	Watchdog processor checking the sequence of checkpoints (labels) reached by the controlled processor.	Watchdog processor.
[114], [115], [116].	Hardware faults (mostly transient), control flow faults.	Duplication of all high-level language statements, variables, function arguments etc. Control flow signatures.	No
SPCD [117].	Hardware faults (mostly transient).	Duplication of all high-level language statements in a procedure, or of the procedure call.	No
EDDI [95], SWIFT [118].	Hardware faults (mostly transient).	Duplication of all assembly instructions (and data memory in EDDI).	No

Continued on Next Page...

Table 2.3 – Continued

Technique	Verification target	Method	Hardware support
CRAFT [119].	Hardware faults (mostly transient).	Duplication of all assembly instructions (based on SWIFT [118]). Hardware support to check load and store instructions.	Yes
ED^{4}I [120].	Hardware faults (also permanent).	Duplication of the whole program (executing twice). In the duplicate program all variables are multiplied by a factor of k.	No
[121].	Hardware faults in memory.	ECC in memory blocks, periodic scrubbing.	No
[122].	Burst hardware faults in memory.	Duplication of the whole application.	No
[123].	Illegal memory references.	Watchdog processor performs capability checking (if a memory request has sufficient permissions).	Watchdog processor.
[124].	Application crashes or hangs.	Watchdog module sending "alive report" requests, checkpointing.	No

2.5.1 Approaches Targeting Software Design Faults

Approaches targeting software design faults are based on redundancy in the form of multiple modules performing the same function, but exploiting as much design diversity as possible. The *recovery blocks* approach [100] divides a software application into multiple modules, possibly with several levels of nested modules. Each module ("recovery block") has an alternative module which performs the same function, but is implemented independently to ensure design diversity. The independent implementation of the alternative module increases the probability that the modules will not fail in the same way.

After finishing execution, each recovery block runs acceptance test(s) to verify correctness of the results. The acceptance tests are not required to provide an absolute correctness guarantee, they use approximation. This is because the complexity of exhaustive tests can easily become comparable to the complexity of the tested code. If the acceptance test is passed, the execution progresses to the following module. Otherwise, if a fault is detected, an alternate attempts to perform the function. If all the alternates of a recovery block fail, the higher-level recovery block is declared to fail, and an alternative higher-level recovery block is executed. This scheme is analogous to the standby sparing in hardware (see Section 2.3).

N-version programming [50] creates multiple versions of a software application, preferably by different developer teams, using different programming languages. The results produced by the different versions are compared against each other. This scheme corresponds to the N-Modular Redundancy in hardware.

2.5.2 Approaches Targeting Hardware Faults

Many pure software techniques detecting hardware faults have been proposed. Although they typically introduce a larger performance overhead than pure hardware and hardware-assisted software (hybrid) techniques, they are cheaper to implement. No hardware modifications are required, thus the cost and design time are reduced.

Several software schemes for generating checkpoints have been proposed. Checkpointing refers to saving the active state of a running process so that, upon error detection, the process can be rolled back to the known good state and its execution can be restarted from there. The major problem facing such schemes is checkpoints distribution, i.e., finding optimal locations for checkpoints. If checkpoints are generated very often, it will introduce a huge performance penalty. On the other hand, the fewer checkpoints are inserted, the more computations are lost on a rollback, thus the recovery overhead increases. Li and Fuchs [101] introduce Compiler-Assisted Techniques for Checkpointing (CATCH). A compiler extends an application with instructions that examine the clock value and, if a sufficient time passed since the previous checkpoint, establish a new checkpoint. Long et al. [102] propose a static compiler scheme, in which profiling results are used to choose the appropriate locations for checkpoints.

The Signatured Instruction Stream (SIS) technique [103] proposed by Schuette

and Shen checks the correct sequencing of executed instructions. A program is partitioned into blocks in which there exists only a single valid path from the entry to the exit point. A compiler generates the instruction sequence signature for each valid path and embeds it into the exit point. The same signatures are generated at runtime and compared with the expected ones. A mismatch signals an error. To minimize the memory overhead introduced by storing signatures, the Branch Address Hashing (BAH) technique is used, which encodes a block's signature into the target address of a branch instruction at the exit point (if it exists).

Miremadi et al. [109] describe two software fault detection techniques examining application control flow. Block signature Self-Checking (BSSC) partitions an application code into basic blocks without outgoing (conditional jumps) and incoming (jump targets) branches in it. Every basic block is extended with instructions setting the current signature in the beginning and checking this signature at the exit of the block. A wrong signature signals a control flow error. In addition, the Error Capturing Instructions (ECI) technique inserts instructions triggering an error signal into memory locations which are never executed in normal conditions. A jump to such a location identifies a control flow error. The authors propose to use these techniques in combination with a watchdog timer to enlarge the error coverage. A watchdog timer can detect the cases when a CPU is unable to continue execution, enters infinite loop, etc.

McFearin and Nair [104] introduce a similar technique called Control-flow Checking using Assertions (CCA). An application is partitioned into Branch Free Intervals (BFIs), each with a unique Branch free interval IDentifier (BID). In addition, every BFI keeps a Control Flow IDentifier (CFID), which is identical for the leaves of a common parent. When execution enters a BFI, the new BID is saved in a special variable, and the new CFID is enqueued in a two-element queue. When execution exits a BFI, the current BID is tested, and the previous CFID is retrieved from the queue and verified. CCA is later extended with Enhanced CCA (ECCA) [105]. In ECCA, several BIDs can be grouped into a single block. The more BIDs are incorporated in a single block, the less overhead is introduced, and the longer the fault detection latency is. In addition, the number of instructions used in the entry and exit points of each block is reduced compared to CCA.

Huang and Kintala [124] introduce three software modules that can be integrated in applications to enhance their FT. The modules detect faults in a monitored process and restart it, periodically checkpoint critical state information, and replicate and synchronize files used by the process. The scheme detects

when an application crashes or hangs, and is based on the watchdog principle: a module periodically sends "alive report" requests to the checked application. The modules can be used in Unix software and require a minimal programming integration effort.

Rebaudengo et al. [114, 115] propose a high-level language redundancy method. Every variable has a shadow copy which must always be consistent with the original. A set of rules is defined for every type of a language construct. For example, each assignment statement is executed for both the original and the shadow variables, and the result is compared. All arguments provided to a function are duplicated, as well as the function return value. In addition, to verify the control flow, the code is partitioned into branch-free basic blocks similar to [109]. The first instructions in each basic block assign its associated signature to a special variable, which is checked at the exit from the block. Every branch statement is followed by a check determining if its condition was tested correctly. Every procedure is assigned an identifier, which is written to a special variable. After a procedure returns, its identifier is checked. To guarantee a proper compilation, all the compiler optimizations must be disabled when processing the source code instrumented with this method. Preliminary experimental results demonstrate that the technique increases the executable code size by a factor of 2, and introduces a performance overhead by a factor of 5.

Shirvani et al. [121] consider using error detection and/or correction codes for memory blocks in software. A direct mapping of a similar hardware method, when every word written to memory is encoded and every word read from memory is decoded, is not considered feasible for software, since it introduces a very substantial overhead. To avoid this, the authors propose to protect (encode) only read-only memory parts, such as code segments. A periodic scrubbing is performed to detect and/or correct possible errors. In addition, a special interface can be devised which the protected software can use to encode/decode the necessary memory parts at appropriate time. When protection of a certain memory area is requested, the error detecting and correcting program receives its address and size, encodes the data and stores the check bits in a separately allocated memory region.

Error Detection by Duplicated Instructions (EDDI) [95] duplicates all instructions in the program assembly code and inserts checks to determine if the original instruction and its duplicate produce the same result. More precisely, the registers are partitioned into two groups, one for the original instructions and one, called the shadow registers, for the duplicate instructions. After the execu-

tion of a duplicate instruction, the contents of the shadow register(s) it affects should be identical to the contents of the original destination register(s). A mismatch signals an error. Instead of comparing the registers after the execution of every duplicate instruction, EDDI allows faults to propagate until the point where the value is saved to memory, and detects them just before saving. In other words, EDDI compares the registers only before their values are stored in memory. This minimizes the number of checking instructions needed, and thus, reduces the performance penalty, while data integrity is still guaranteed. EDDI also duplicates the data memory. This means that the data memory has a shadow copy which is referenced by the duplicate load/store instructions. Thus, after any duplicate store instruction, the contents of the shadow data memory must be the same as the original data memory. From the performance point of view, the main idea behind EDDI is that most applications cannot fully exploit wide-issue superscalar processors because they do not exhibit sufficient ILP [125]. Because the original instructions and the duplicate instructions are independent, applying EDDI will increase ILP and, therefore, detect errors with a minimal or reasonable performance penalty in superscalar processors. Experimental results demonstrate a performance overhead of 13% to 106% on a 4-way superscalar processor.

Oh and McCluskey [117] introduce a technique called Selective Procedure Call Duplication (SPCD), which minimizes the energy consumption overhead of protective redundancy. For every procedure, SPCD either duplicates all its statements in the high-level language source code (similar to [114, 115]), or duplicates the call to the whole procedure, comparing the results. The decision is based on the error latency constraints imposed. A heuristic algorithm using the program call graph is proposed for that. If the latency of calling a procedure twice and checking the results is lower than the accepted error latency, the whole procedure call can be duplicated instead of every statement in it. Procedure-level duplication is a coarse-grain version of the fine-grain statement-level duplication. It reduces the energy and performance overhead by decreasing the number of checks executed. Procedure-level duplication also substantially reduces the code size compared to the statement-level duplication, because the same procedure is called twice instead of duplicating every static instruction in it. Experimental results report a maximum energy savings of 26.2% for SPCD compared to EDDI [95] and the instruction duplication approach in VLIW architectures presented in [64].

Oh et al. [106] introduce the Control-Flow Checking by Software Signatures (CFCSS). Similar to [109], this is a pure software control flow error detection technique which does not require a watchdog processor or multitasking. Each

node in the program graph has a unique signature, which is embedded into the compiled application together with error detection instructions. When executing, the embedded signatures are compared to the ones generated at run-time.

Oh et al. [120] propose a technique called Error Detection by Diverse Data and Duplicated Instructions (ED^{4}I). ED^{4}I implements instruction duplication at the high-level language level by copying a whole program. The original and duplicate programs compare the results after execution. To address permanent faults, when creating the duplicate program, all the variables are changed (multiplied by a factor of k). When comparing the results, the appropriate adjustments are made to the result of the duplicate. This ensures that permanent faults affect the redundant program copies in different ways, producing different (detectable) results. Optimization of the k factor improving the fault coverage of ED^{4}I is discussed in detail.

A control flow verification technique called Assertions for Control Flow Checking (ACFC) is proposed by Venkatasubramanian et al. [107]. ACFC adds special instructions to every application basic block. These instructions calculate the execution parity of the block. When the precomputed parity does not match the parity calculated at run time, an error is detected. A preprocessor has been developed that instruments a C program with the necessary assertions. Another assertion-based control flow verification technique called Yet Another Control-Flow Checking Using Assertions (YACCA), is proposed by Goloubeva et al. [108]. Nicolescu and Velazco [116] propose an error detection scheme using a high-level language variables redundancy, similar to [114,115], and a control flow verification method similar to others discussed above.

Reis et al. [118] combine EDDI [95] with CFCSS [106], producing a technique called Software Implemented Fault Tolerance (SWIFT). Unlike EDDI, SWIFT does not replicate memory, but relies on ECC or other appropriate memory protection methods. SWIFT improves the fault coverage and performance of the previously proposed techniques. A SWIFT-based hybrid (involving hardware modifications) technique named CompileR-Assisted Fault Tolerance (CRAFT) [119] increases reliability and performance of the pure software techniques.

Saha [122] proposes a software-only technique capable of detecting and tolerating burst memory errors. It uses two or more copies of an enhanced application code. A performance penalty of about 2.7 times is introduced for the error-free scenario.

2.5.3 Software Techniques Using Watchdog Processors

Some SIFT techniques involving additional processors (watchdog processors) are presented in this section. They are not purely software, but hybrid methods, involving both software and hardware units called watchdog processors. Watchdog processors concurrently check some aspects of the functionality of the main processor [126]. For example, possessing the correct control flow information of the application running on the main processor, a watchdog processor can compare the actual control flow with it, declaring an error on any deviation. The effect of a carefully designed watchdog processor to the fault detection in a system is similar to the effect of a full replication. Watchdog processors are typically significantly smaller and cheaper than the processors which they control, and enjoy several other advantages. Because the design of watchdog processors usually differs from the design of the checked hardware, the watchdog concept provides design diversity [13, 127], and has the potential to detect not only hardware, but also software and design errors.

One of the early works on this topic is presented in [110]. Using a checking program running on a watchdog processor (called *observer*), the control structure of the application running on the main processor (*worker*) is verified at run-time. Certain checkpoints are embedded into the worker. While the worker runs, it sends the checkpoint information to the observer, which checks if the sequence composed by the previous and the current checkpoints is valid. For example, if every procedure has a checkpoint, the correctness of the procedures sequence can be verified by the observer.

The use of a watchdog processor is recommended by Yau and Chen for the concurrent control flow checking approach presented in [111]. An application is partitioned into basic blocks. The path information for each of these blocks is available for the checker, which compares it with information delivered at run-time. In this manner, any error resulting in illegal branch in a loop-free block is detected.

In Structural Integrity Checking (SIC) proposed by Lu [112], the controlled application incorporates *labels* which are sent to the checker running on a watchdog processor. The checker program is structurally similar to the checked one, has the same labels in appropriate control paths, but does not include the actual computations (that is why it requires less computational resources). While running, the controlled application sends to the checker the labels in the order as they appear. In the absence of faults, the labels sent by the controlled application must match those in the checker. Because the checker does not compute variables which determine the control flow, when coming to condi-

tional branches, it does not know which path has to be followed, and can only check if one of the possible paths is taken (but it is possible to extend SIC with conditional branch prediction). In addition, SIC makes sure that a return corresponds to every function call. In SIC, a programmer does not need to provide any abstract description of the application. The programs running on the main processor and watchdog processor are directly compiled from the same source code, using a special compiler preprocessor.

Namjoo and McCluskey [123] propose a capability checking watchdog processor. This low-cost processor catches illegal memory references issued by the main processor, by performing capability checking. Capability checking references to testing if a code object (for example, a program) issuing a memory request has appropriate access permissions for the object being targeted.

In the approach of Michel et al. [113], a watchdog processor checks the sequence of the control flow graph nodes separately from the instruction stream inside each node. The main processor provides the watchdog with information about the current node executed. The watchdog verifies the correctness of the execution path, and computes and checks the instruction stream signatures for instructions in the current node.

Mahmood and McCluskey provide a detailed survey and comparison of FT techniques based on the use of watchdog processors existing by 1988 in [126].

2.6 FT Techniques for Cache Coherence

Current trends in computer architecture demonstrate an increasing interest in multiprocessor systems [128]. Most of the FT techniques discussed so far are also applicable to multiprocessors. However, due to some unique features not present in uniprocessor systems, multiprocessors introduce a new direction for FT research. This section discusses recent work on FT in a multiprocessor-specific area, cache coherence, whose correctness is essential for systems employing it.

Multiprocessors feature a distributed and/or shared memory hierarchy. To enhance programmability, the details of the memory structure are often hidden from the application developer, for whom the memory structure appears as a single shared memory accessible from all the nodes. This is achieved by employing cache coherence protocols which guarantee a consistent memory view for different nodes [125]. Cache coherence plays a very important role in multiprocessors, providing data integrity. For example, it guarantees that, when

one processor has changed data at a certain memory address, other processors in the system fetch a valid copy of these data when they read them. Thus, faulty cache coherence hardware can lead to data integrity violation. In the above example, if the cache coherence hardware fails to notify a reading processor about the changes another processor has made to the data, it will work with wrong (stale) data, produce other wrong data on this basis, etc.

Several dynamic cache coherence verification techniques have been proposed. All of them introduce a certain hardware overhead, and most of them increase the communication network bandwidth requirements.

In one of the earliest works on this topic, Cantin et al. [129] propose a method based on the watchdog processor concept. A checker circuit is placed on each cache and implements a simplified version of the coherence protocol employed (without the additional optimizing states to reduce the complexity). The checker maintains its own copy of the tags. When the state of a cache line changes, the necessary information is sent to the checker. The checker recomputes the coherence transaction and verifies the states produced by the cache. A mismatch signals an error. In addition to local verification, the checker performs global verification. The new state is broadcast to the checkers of other caches in the system, using a special logical network (which can be a separate or the existing system network). The other checkers make sure that the new state of the broadcasting cache does not conflict with their own states. For example, multiple caches should not have the same line in the modified state, or in modified and shared states.

Meixner and Sorin [130] develop the Token Coherence Signature Checker (TCSC) scheme. TCSC is targeted at memory systems utilizing Token Coherence [131], but can also be applied to the traditional invalidation-based snooping and directory coherence protocols, if they are interpreted in terms of token coherence. TCSC keeps two signatures on every cache and memory controller. One signature represents the history of coherence states for all the blocks in cache or memory, and the other incorporates the history of data values stored. Periodically these signatures are sent to the verifier(s). Using the coherence states signatures, the verifier determines if any conflicting coherence states appeared in different places and if data propagated incorrectly. To apply TCSC to a traditional coherence protocol, the MOSI protocol was interpreted in terms of token coherence. TCSC requires additional information to be included in the coherence messages and introduces extra traffic by submitting the signatures to the verifier. Implementing the snooping MOSI protocol requires more additional messages. TCSC introduces a theoretical

worst-case bandwidth overhead of 4.54% for the Token Coherence, 10% to 5% (with some optimizations) for snooping-based protocols, and 15% to 10% for directory protocols. TCSC also requires additional hardware to compute the signatures and for the verifier(s).

In a work prior to TCSC, Sorin et al. [132] compute signatures locally and submit them periodically to a central verifier, similar to TCSC. They use a different algorithm for computing the signatures and have different signatures for the coherence and messages history. The technique is limited to snooping systems and provides slightly less fault coverage than TCSC, but requires less network bandwidth.

This dissertation (Chapter 6, also [133]) proposes to use independent checkers in snooping systems. Every cache has associated checker(s) that keep the current state information for the cache lines. By snooping the network traffic sent to and from the checked caches, the checkers maintain their cache line states and detect almost all coherence errors in the checked caches, as well as some network errors. The only modification of the original system required is the addition of a few bits to every network message sent by the checked cache (3 additional bits in the case of the MESI coherence protocol and 2-way set-associative caches). These bits pass to the checkers some necessary information, such as the current coherence state of the cache line in the checked cache, and the way at which the cache line is kept in associative cache. Further the checkers work completely autonomously when connected to the system snooping network.

Fernandez-Pascual et al. [134] propose a fault tolerant token coherence protocol capable of tolerating transient errors in CMP interconnection networks. For example, message loss is handled by the protocol. This is achieved by running several different timers that start the *token recreation* process when a corresponding timeout is detected. In comparison with a non-fault tolerant token coherence protocol, the proposed protocol incurs a negligible performance penalty in the fault-free scenario, and about 15% overhead when up to 250 messages per million are lost.

2.7 Summary

This chapter has introduced the FT in computing systems. A brief history of FT development and conventional FT approaches have been presented. Then POFT techniques for different system organizations, such as VLIW and superscalar, have been discussed. Software FT approaches, targeting both software

design faults and hardware faults, have been introduced. Finally, FT techniques to verify the cache coherence operation in multiprocessor systems have been presented.

In the following chapters several novel POFT approaches and techniques are proposed, mostly targeting the superscalar organization. In addition, Chapter 6 proposes a cache coherence verification method which outperforms existing methods at the expense of a slightly reduced fault coverage.

3

Instruction-Level Fault Tolerance Configurability

This chapter introduces an approach to FT called *Instruction-Level Configurability of Fault Tolerance* (*ILCOFT*). ILCOFT is based on the observation that different instructions have various effect on the reliability of an application. Undetected faults in some instructions can cause the whole application to crash. Undetected faults in other instructions, however, only slightly corrupt the final application output, or are even invisible in the output. This observation indicates that the application developer should be able to assign various protection levels to individual instructions. ILCOFT provides the programmer with such a capability. ILCOFT can be applied to many existing FT techniques, thereby reducing their performance and energy consumption overhead, and/or improving their reliability.

This chapter is organized as follows. Section 3.1 motivates the ILCOFT approach. Section 3.2 presents ILCOFT. Section 3.3 presents and analyzes experimental results at the kernel level, and Section 3.4 at the application level. Finally, Section 3.5 summarizes this chapter.

3.1 Introduction

In existing FT techniques, there is always a trade-off between FT and cost, either in performance or resources. System hardware resources are limited, and the more of them are dedicated to FT, the more performance suffers. Furthermore, the redundancy introduced to provide FT dissipates additional energy. It is therefore desirable to have a configurable system which is able to use its resources to improve either FT or performance. Some proposed FT schemes enable system configuration before an application is run, which al-

lows to choose between higher performance or stronger FT depending on the application requirements.

Saxena and McCluskey [91] notice that multithreaded FT approaches can target both high-performance and high-reliability goals, if they allow configuration to either high-throughput or fault tolerant modes, as is the case for slipstream processors [85,86]. Breuer, Gupta, and Mak [135] propose an approach called Error Tolerance, which increases the fabrication yield. This is achieved by accepting fabricated dies which are not completely error-free, but deliver acceptable results. The tolerance of multimedia applications to certain errors is discussed in this context. Chung and Ortega [136] develop a design and test scheme for the motion estimation process. This reveals that the effective yield can be improved if some faulty chips are accepted. Reis et al. [118] present a software fault detection scheme Software Implemented Fault Tolerance (SWIFT) which duplicates instructions, compares their results at strategic places, and checks the control flow. The authors mention that SWIFT can allow a programmer to protect different code segments to varying degrees, like ILCOFT does. Lu [112] presents the Structural Integrity Checking technique using a watchdog processor [126] to verify the correctness of an application control flow. "Labels" are inserted into the application at the places where a check should be performed. The higher the density of the "labels", the more checking is done. Thus, a programmer can increase the density of the "labels" at the critical parts of an application, increasing the amount of checking applied to them.

We propose to leverage the natural error tolerance of certain applications (such as multimedia) to improve their overall reliability and/or improve their performance and resource consumption. This goal can be achieved by a system which can be configured to target either FT or performance at the instruction, rather than the application, level. A developer should be able to configure the strength of FT techniques applied to particular instructions or blocks of instructions in the application. This is useful for applications in which more and less critical parts (instructions) can be distinguished. For example, as noticed in [135, 136], for multimedia applications, most of the computations do not strictly require absolute correctness. Many errors in these computations would not be noticeable for a human, while others can cause a slight but tolerable inconvenience. The application parts performing these computations can have lower or no protection with little risk. This minimizes protective redundancy which degrades performance and/or increases the system cost. However, other parts of the same applications can be very critical. For example, if the control of a multimedia application is damaged, the whole application is likely

to crash. As another example, a fault in the data assignment to the quantization scale can affect the quality of the video significantly. These parts require strong FT. Moreover, the time and/or resources saved by reducing the redundancy for non-critical parts can be used to enhance the FT of the critical parts even further. In this case, the overall reliability of the application increases, at the expense of reduced reliability of non-critical parts. By reducing the protection of non-critical and increasing it for critical application parts, a developer can trade-off resources and reliability, fine-tuning it for the particular purposes.

We call the strength of FT features applied to an instruction the *degree of FT*. The more efficient FT techniques are applied, the higher the degree of FT is. The minimum degree of FT corresponds to the absence of any FT techniques. Duplication and comparison of the results has a lower degree of FT than TMR [3, 31]. A higher degree of FT corresponds to a larger amount of redundancy, and hence, is more expensive in terms of resources and/or time.

To enable ILCOFT, a system should support several degrees of FT. An application developer is able to specify the desired degree of FT for each instruction or group of instructions. This can be done either in high-level language or in assembly code. Partially, it could also be performed automatically by the compiler. The system applies one of the existing FT schemes to satisfy the needs of particular instructions, for example, by duplicating or triplicating them in software or hardware, and possibly comparing the results.

3.2 ILCOFT

This section presents *Instruction-Level Configurability of Fault Tolerance* (*ILCOFT*). ILCOFT allows to apply different degrees of FT to different application parts, depending on how critical they are. ILCOFT is a general technique that can be applied to many existing FT schemes, as will be shown in Section 3.2.3. A particular ILCOFT implementation depends on the system architecture (the FT scheme employed), and the application constraints. ILCOFT can be applied both to hardware as well as software FT schemes. A certain hardware support is required in the case of hardware FT schemes. Moreover, ILCOFT always requires a certain software-level activity to assign degrees of FT to application parts, as will be discussed in Section 3.2.2. The actions taken by the system in the case of a fault detection depend on the FT scheme adapted. For example, FT schemes providing only error detection will terminate the faulty execution unit, possibly performing a graceful degradation, etc. FT schemes supporting recovery may recover and continue execution. The FT

characteristics of ILCOFT-enabled FT schemes depend on the FT techniques which are adapted and on the developer's instructions ranking in terms of their criticality.

Section 3.2.1 gives the reasoning behind ILCOFT. Section 3.2.2 discusses possible ways for an application developer to specify the required degree of FT for particular instructions or code blocks. Finally, Section 3.2.3 shows how several existing FT schemes can be adapted to support ILCOFT.

3.2.1 Motivation

Many application domains, such as multimedia applications and software-defined radio [137], are naturally tolerant to some faults. Some errors are unavoidable in these domains, and introducing a limited number of additional similar errors does not seriously affect the operation. For example, many multimedia applications, such as image, video and audio coders/decoders, use lossy algorithms. After decoding, the stream produced is not perfect. It incorporates errors which the human eye cannot notice or can easily tolerate. For example, if one of more than 307 thousand (640×480) pixels in an image or a video frame has a wrong color, it is likely to be ignored by a human. Furthermore, if an error occurs in calculations associated with motion compensation in video decoding, it can result in a wrong (rather small) block for one or a few frames. The number of frames that can be affected depends on the place where the error appeared and on how far the following key frame is. Because usually there are 20 to 30 frames per second, the chance that a human will notice this error is quite low. Moreover, if it is noticed, it will probably result in less inconvenience than compression-related imperfections. Errors can be allowed in this kind of computations. If, however, an error occurs in the control part of a multimedia application, it is very likely that the whole application will crash. Furthermore, errors in some other parts can lead to a significant output corruption. Therefore, errors are not allowed to occur in the latter cases.

To illustrate critical and non-critical instructions, consider the image addition kernel presented in Figure 3.1. If an error occurs in any of the expressions that evaluate the pixel value *sum*, it will result in a wrong pixel in the output image; this is tolerable. If, however, a problem appears in the statements controlling the loops, there is a very small chance that it will not crash the application or seriously damage the results. A normal termination with correct results can happen in this case if one or both loops performed too many iterations, but the memory which they damaged was not used (read) later. This scenario,

however, has a very low probability. It is likely that the application will crash (due to a jump to an invalid address, invalid memory contents, etc.), or, if the loop is exited too early, the part of the image which has not been processed yet will be wrong. The *if* statement which controls saturation is less critical than the loops, because if the condition is evaluated incorrectly, only one pixel is affected. If the branch target address is corrupted, however, the application will most probably crash. Thus, this *if* statement can also be considered for a higher degree of FT.

```
for( i=0; i<N; i++ )
    for( j=0; j<M; j++ )
    {
        sum = ImageX[i][j] + ImageY[i][j] ;
        if( sum > 255 )  /* saturation */
            sum = 255;
        ImageX[i][j] = sum;
    }
```

Figure 3.1: Image addition.

In ILCOFT, the programmer specifies the required FT degree of every instruction or group of instructions. In other words, the programmer indicates which parts of an application are critical and which are not. In Section 3.2.2 we describe how this can be done by the programmer, and under which circumstances it can be performed automatically by the compiler. For example, for the image addition kernel presented in Figure 3.1, the programmer should specify the maximum FT degree for the instructions controlling the loops and the branch target address of the *if* statement. For the other instructions, which calculate the pixel values, a lower FT degree is acceptable, and even desirable, when aiming at performance and resource consumption minimization.

By reducing the degree of FT of non-critical instructions, ILCOFT reduces the required time or resource redundancy to implement FT. Space redundancy, which increases the amount of required hardware and energy resources, can achieve FT without a performance loss, but at the expense of increased resources cost. The amount of hardware is often limited, however, and to achieve FT under this constraint, time redundancy is used, which degrades performance, and keeps energy consumption high. When both resources and time are limited, which is very common, ILCOFT increases the performance and reduces the energy consumption at the expense of decreased reliability of non-

critical application parts. The critical parts, however, are still as reliable as with a full FT scheme, so the overall application reliability is not affected. Optionally, the time saved can be used to further improve the FT of the critical application parts by applying more time-redundant techniques to them. In this case the overall application reliability increases, because its critical parts are protected better.

3.2.2 Specification of the Required FT Degree

Two possible ways to specify the desired degree of FT applied to an instruction are to set it in assembly code or in high-level language. Alternatively, the compiler can perform this automatically.

We do not consider it feasible for large applications that a programmer marks the required degree of FT for every assembly instruction or high-level language statement manually. It makes sense to first choose the appropriate policy which determines the default degree of FT. The default degree of FT is applied automatically to all unmarked instructions. It can be set to, for example, the minimum, maximum, or average possible degree of FT, as will be explained below.

The approach which sets the default degree of FT to the minimum requires a programmer to mark instructions/statements that should receive a higher degree of FT. This method does not look very practical, because there is a high chance that many instructions are critical for an application, e.g., an illegal branch in any place can crash the whole application.

The opposite approach, when the default degree of FT is the maximum, looks more useful for many applications. In this case, the programmer marks the instructions or statements that should have a lower degree of FT, and all others get a higher degree. This is especially suitable for multimedia applications, many of which spend most of the runtime in small kernels. Decreasing the degree of FT of a few computational instructions in the most time consuming kernels can provide a significant application-level performance gains, as we will demonstrate in Section 3.4.

Finally, the default degree of FT can be assigned some intermediate value. Then, a programmer has to specify instructions/statements requiring higher and lower degree of FT.

Next we discuss how an application developer can specify the degree of FT

in the assembly or high-level language source code, and how it can be done automatically by the compiler.

In Assembly Code

If the developer specifies the required degree of FT in assembly code, the way it can be done depends on the employed FT scheme, if it is a hardware or software technique.

If FT is implemented in hardware, the way the programmer marks instructions might depend on how the degree of FT is passed to the hardware (see Section 3.2.3). When the degree of FT is embedded in the instruction encoding, the programmer marks instructions using some flags, and the assembler encodes the necessary information into every instruction. With special FT mode configuration instructions, the programmer places these instructions in appropriate places. With separate versions of every instruction, the programmer uses an appropriate version. Alternatively, the assembler can be designed to support hardware-independent marking, which is translated automatically into the supported FT degree communication scheme. Then a programmer always marks instructions in the same way.

In software, the EDDI technique [95] discussed in Section 3.2.3 can be employed. Adapting EDDI, a programmer can duplicate the critical instructions manually, taking care about the register allocation, register spilling, possibly memory duplication, etc. However, an automatic assisting tool would be very useful. This tool can be based on the compiler postprocessor used in [95], which automatically includes EDDI into an application. The compiler reserves registers for duplicate instructions, and the tool duplicates everything. In the resulting assembly file, the programmer removes the undesired redundancy manually.

In High-Level Language

Figure 3.2 depicts how programmers could specify the desired degree of FT for particular statements or blocks of statements in a high-level language. This is done in the form of a *#pragma* statement which determines the degree of FT that should be applied to the following statements, until the next *#pragma* statement changes it. The larger the number corresponding to *FT_DEGREE* is, the higher the degree of FT should be. Each statement is compiled into instruction(s) whose degree of FT is equal to that of the corresponding state-

ment. In the case of control statements, the compiler must be able to find their dependencies and to apply the appropriate degree of FT to them. To be on the safe side, by appropriate degree of FT here we mean the highest between the previously assigned degree and the one required for the considered control statements.

In Figure 3.2, the instructions which are generated for the *for* statement, should have the degree of FT equal to 3. The instructions inside the loop (and after the loop until the next *#pragma*) should have the degree of FT 1. Obviously, the loop control depends on the values of the variables *i* and *n*, which have been assigned before. Hence, the compiler should walk backwards to find all the instructions on which the values of these variables depend, and assign the degree of FT 3 to them.

```
#pragma FT_DEGREE 3

for( ; i < n; i++ )
{
#pragma FT_DEGREE 1

      c[i] = a[i] + b[i];
}
```

Figure 3.2: Possible FT degree specification in a high-level language.

Automatically by the Compiler

If a system supports only two degrees of FT, for example, *no FT* (no FT techniques are applied) and *fault tolerant* (some techniques are applied), in some cases the compiler can determine the instructions that need to be fault tolerant automatically. This saves the programmer from manual work. The automatic compiler scheme can be based, for example, on the observation that in most cases, the instructions on which an application's control flow depends, require a higher degree of FT. All control flow instructions, such as branches, jumps, and function calls, are assigned a higher degree of FT. Furthermore, all instructions on which these control flow instructions depend should also receive the higher FT degree. The efficiency of this scheme depends on the compiler's ability to perform exact dependence analysis. In the worst case, all instructions on which a control flow instruction could depend need to be given the higher

FT degree.

3.2.3 FT Schemes Adaptable to ILCOFT

Fault tolerant systems adapted to support ILCOFT need to provide several FT techniques of varying strengths, corresponding to different degrees of FT. For example, a non-redundant instruction execution has FT degree 0 (no FT), duplication with comparison of the results can be assigned FT degree 1, and a Triple Modular Redundancy (TMR) is associated with FT degree 2. Duplication and triplication assumes either hardware or time redundancy. Hardware redundancy can take the form of multiple execution units where the copies are executed simultaneously. Time redundancy is provided by a sequential (or partially sequential) execution of multiple copies.

Because the main goal of ILCOFT is to optimize performance and energy consumption, we focus on FT techniques with similar objectives. ILCOFT does not target systems for which only a high level of FT is important, and a large amount of redundancy is not an issue. There exist several techniques for high-performance processors that try to minimize the performance overhead created by protective redundancy. Below we discuss how some of them can be adapted to support ILCOFT.

The EDDI (see Chapter 2, Section 2.5.2) instruction duplication scheme supports only one degree of FT: duplication and comparison. It is straightforward, however, to extend EDDI to allow more redundancy, by implementing, for example, TMR. A lower degree of FT (no redundancy) can be easily achieved by avoiding duplication of certain instructions. From now on we assume that the user can specify the degree of replication.

In ILCOFT-enabled EDDI, only critical instructions are replicated. As discussed in Section 3.2.2, the programmer specifies the required FT degree of all program statements or assembly instructions. Alternatively, it is done automatically by the compiler. During compilation, each instruction is replicated according to its FT degree and then the results are compared or voted. Memory replication is not used in ILCOFT-enabled EDDI because all instructions have to be replicated to maintain a consistent memory copy. Instead, memory protection can be implemented by using ECC or other popular methods, preferably in hardware. In Section 3.3 and Section 3.4, the performance and energy dissipation of ILCOFT-enabled EDDI is compared to those of EDDI. It will be shown that minimizing the degree of FT for non-critical instructions

provides a substantial gain. In addition, the fault coverage of these schemes will be evaluated.

Franklin [69] proposed to duplicate instructions in superscalar processors at run time and compare the results to detect errors (see Chapter 2). To adapt this scheme to support ILCOFT, the required FT degree of executed instructions has to be passed to the hardware. Based on this information, the hardware performs the appropriate FT action, i.e., duplicates the instructions if necessary. This can be also applied to the scheme proposed in [74].

The DIVA approach [71, 88, 90] uses a simple and robust processor, called DIVA checker, to verify the operation of the high-performance speculative core. This approach can also be adapted to support ILCOFT by selecting the instructions whose results have to be verified by the DIVA checker.

ILCOFT is also applicable to FT techniques based on simultaneous multithreading [96], such as those presented in [81–83, 91, 98], slipstream processors [85], and others (see Chapter 2).

It should be noted that for hardware FT techniques, there must be a way to set the required FT mode for every instruction. For example, several bits in the instruction encoding can specify the required FT degree. The number of bits allocated for this purpose depends on the number of available FT modes supported by hardware. Alternatively, special instructions can be introduced which configure the hardware to work in the desired FT degree mode. Finally, separate versions of each instruction can be created for every supported FT degree. The last solution does not look promising, however, because it implies a large overhead.

3.3 Kernel-Level Validation

This section presents experimental results for several kernels. The advantages provided by applying ILCOFT to an existing FT scheme are evaluated. Due to the experimental setup limitations, we apply ILCOFT to only one FT scheme, the software error detection technique EDDI [95]. We adapt EDDI to support ILCOFT.

The FT features of EDDI and ILCOFT-enabled EDDI are discussed in Section 3.3.3. ILCOFT-enabled EDDI limits the sphere of replication of EDDI, protecting only the critical instructions, and avoiding memory duplication. Both EDDI and ILCOFT-enabled EDDI reliably protect against transient hardware faults that do not last longer than one instruction execution. To protect

against faults taking more time, including permanent faults, there should be a way to ensure that an instruction and its duplicate execute on different hardware units. For example, they can execute on different CPUs of a multiprocessor system, or on different functional units of a superscalar processor. In the latter case, only long-lasting faults in the functional units are covered. Alternatively, to avoid hardware replication, techniques changing the form of the input operands of the duplicate instruction, such as alternating logic [56] and recomputing with shifted operands [58], can be used. However, these enhancements are expected to have a significant impact on performance.

Four kernels are investigated. *Image Addition* (*IA*), discussed in Section 3.2.1 (Figure 3.1), *Matrix Multiplication* (*MM*) with rather small input matrices of the size 20×10 and 10×20 to reduce the simulation time, and *Sum of Absolute Differences* (*SAD*) used for motion estimation in video codecs, are kernels very often used in multimedia applications. The fourth kernel computes the Fibonacci numbers, which are widely used in science, and even in financial market trading and music [138].

The kernels are chosen to represent a range of algorithms: from (very) tolerant to faults to hardly tolerant to faults. IA contains a large number of independent calculations, which makes it tolerant to faults: a fault in most of the computational instructions can only affect a single output element. MM has many independent as well as many dependent operations. The output elements are, however, computed independently, thus a fault in one of them does not affect the others. Every Fibonacci number depends on all the previously computed numbers, and thus, any fault corrupts all subsequent values. SAD outputs a single value which depends on the whole computation sequence, and is either correct or wrong. Any fault in SAD leads to a wrong result, thus, SAD is the most vulnerable kernel.

This section is organized as follows. Section 3.3.1 presents and analyzes the performance of the considered schemes, Section 3.3.2 the energy results, and Section 3.3.3 the fault coverage.

3.3.1 Performance Evaluation

To evaluate the performance improvement obtained by applying ILCOFT to EDDI, performance results of four kernels in non-redundant (i.e. original), EDDI and ILCOFT-enabled EDDI forms are compared. The sim-outorder

simulator from the SimpleScalar tool set [139, 140] is utilized for performance simulation. The default SimpleScalar PISA architecture (an out-of-order machine with the default issue width of 4, 32 KB L1 data and instruction caches, and 512 KB unified L2 cache) is used.

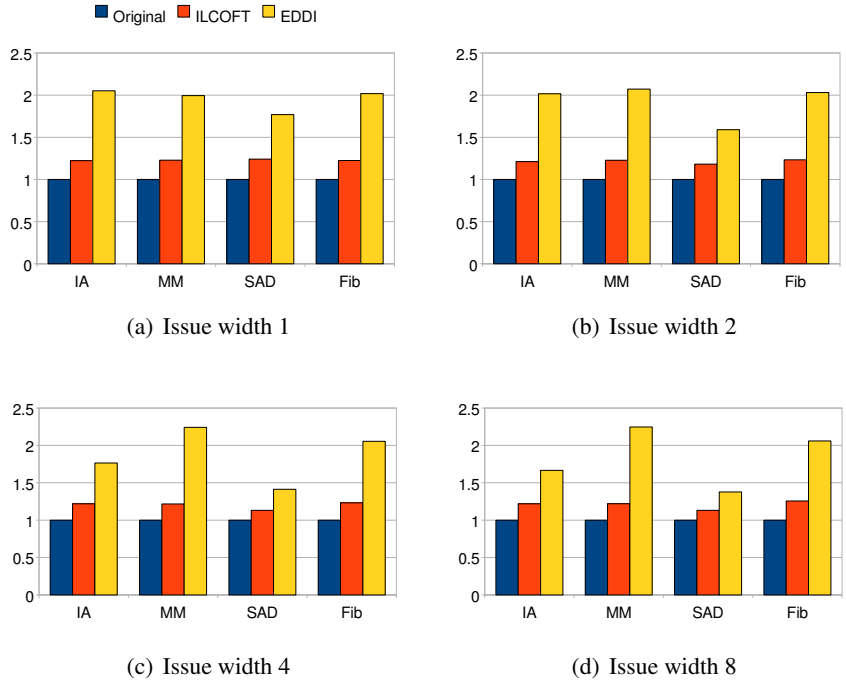
For each kernel, the C source code is compiled to SimpleScalar assembly code. The compiler-optimized version of the application (i.e. compiled by GCC with *-O2* flag) plays the role of the "original", non-redundant application, without FT.

The EDDI version of the kernel is derived from the original version by hand, according to the specification presented in [95]. All instructions and memory structures are duplicated, and the checking instructions are integrated. Checking instructions only appear before a value is stored or used to determine a conditional branch outcome. Faults are free to propagate within intermediate results. This scheme has been proposed in [95] to minimize the performance overhead.

The ILCOFT-enabled EDDI versions are obtained from the original application by duplicating only the critical instructions in the kernel and comparing their results with checking instructions as described above, without memory duplication. The ILCOFT-enabled EDDI versions are also developed by hand. The control instructions are considered to be critical. For the IA kernel, these are the instructions to which the loop control statements in Figure 3.2 are compiled, and the instructions on which the control variables depend.

Figure 3.3 depicts the slowdown of EDDI and ILCOFT-enabled EDDI over the non-redundant scheme for four different processor issue widths. Furthermore, Figure 3.4 shows the ratio of the number of committed instructions of both schemes to that of the non-redundant scheme. Without ILP, speculation etc., Figure 3.3 is expected to be similar to Figure 3.4. Indeed, the performance results of Figure 3.3(a) (issue width of 1) are quite consistent with Figure 3.4, but for larger issue widths, the processor exploits the available parallelism better, since the original instruction and its duplicate are independent. Because of this, the slowdown of EDDI and ILCOFT-enabled EDDI over non-redundant execution decreases when the issue width increases, unless there are other limiting factors. MM, for example, has a structural hazard: there is only one multiplier, so the duplicate of a multiplication instruction cannot be executed in parallel with the base instruction.

Figure 3.3 shows that despite duplication of all instructions and memory in EDDI, especially for larger issue widths, its slowdown over the original application is in most cases smaller than the intuitively anticipated two times (actu-



(a) Issue width 1

(b) Issue width 2

(c) Issue width 4

(d) Issue width 8

Figure 3.3: Slowdown of EDDI and ILCOFT-enabled EDDI versions over the non-redundant version, for varying issue widths.

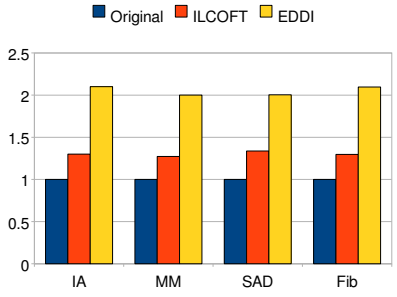


Figure 3.4: Ratio of the number of committed instructions.

ally, more than two because of the checking instructions, duplicated memory, and register spilling). This happens due to the increased ILP introduced by the duplicates which are independent on the original instructions. This leads to a more efficient resource usage and fewer pipeline stalls. ILCOFT-enabled

EDDI also profits from this feature. Figure 3.3 also shows that ILCOFT-enabled EDDI is considerably (up to 50%) faster than EDDI. Several factors contribute to this:

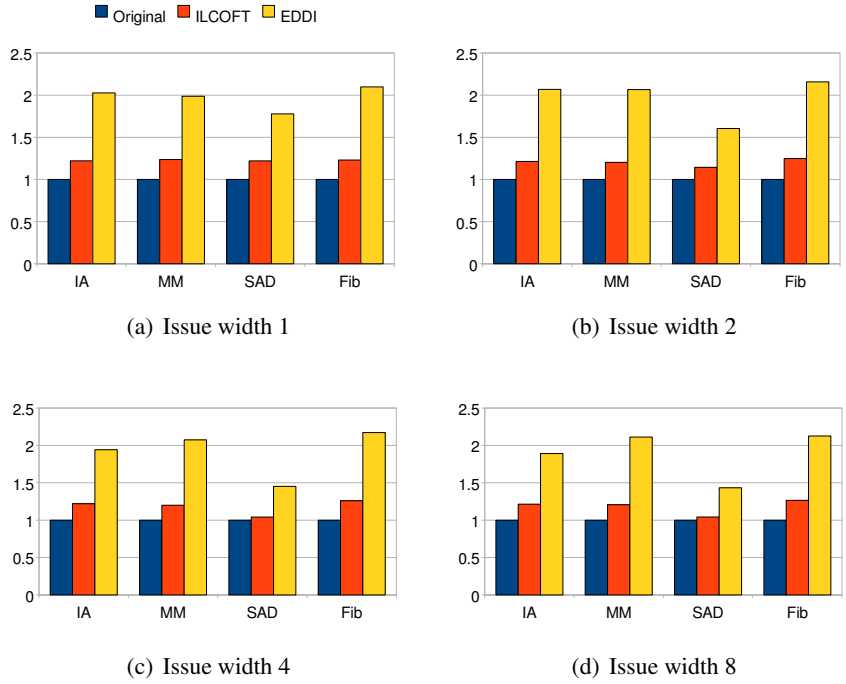
- The number of instructions in ILCOFT-enabled EDDI is smaller than in EDDI (by about 40% on average, see Figure 3.4).
- EDDI duplicates memory, while ILCOFT-enabled EDDI does not.
- EDDI needs more registers than ILCOFT-enabled EDDI, since ILCOFT-enabled EDDI duplicates fewer instructions and, hence, reduces register pressure. Higher register usage leads to more register spilling.

As these factors have different weights for different kernels, the speedup of ILCOFT-enabled EDDI over EDDI is not constant. For example, for the IA kernel, the simulation results show that memory duplication contributes 1.3% to the speedup of ILCOFT-enabled EDDI over EDDI. The contribution of additional register spilling (two more registers are saved on the stack for EDDI) is negligible (less than 1%). The remaining contribution should be attributed to the increased number of instructions.

3.3.2 Energy and Power Consumption

To evaluate the energy saving of ILCOFT-enabled EDDI, we use the power analysis framework Wattch [141]. Wattch is an architectural-level microprocessor power dissipation analyzer. It is a high-performance alternative to lower-level tools which are more accurate, but can only provide power estimates when the layout of the design is available. According to [141], Wattch provides a 1000 times speedup with an accuracy within 10% of the layout-level tools. We use the default Wattch configuration. The results for the clock gating style which assumes that unused units consume 10% of their maximum power dissipation [141] are considered.

The energy consumption increase of EDDI and ILCOFT-enabled EDDI over the non-redundant (original) scheme is presented in Figure 3.5. As can be expected, the energy graphs follow closely the performance graphs of Figure 3.3. This is because the same factors (number of instructions and used resources) affect energy consumption as well as performance. Figure 3.5 demonstrates that ILCOFT is able to significantly (up to 50% with an average of 38%) reduce the energy consumption overhead of the EDDI scheme.



(a) Issue width 1

(b) Issue width 2

(c) Issue width 4

(d) Issue width 8

Figure 3.5: Energy consumption increase of EDDI and ILCOFT-enabled EDDI over non-redundant kernels, for varying issue widths.

The average power consumption per cycle of the different schemes has also been evaluated. The power consumption does not vary significantly for the three considered schemes, because their resource utilization is similar. The maximum variation observed is 10%. As can be expected, the variation is minimal with lower issue widths (no more than 3% for issue width 1), and increases with higher issue widths. This can be explained by approximately equal resource usage with lower issue widths. With higher issue widths, the resource usage varies for different schemes, due to the difference in the available ILP, and the power consumption varies accordingly. In most cases EDDI consumes more power per cycle than the other two schemes, because it exhibits more ILP, and, therefore, keeps more resources busy.

Table 3.1: Fault injection results for the non-redundant scheme for the following kernels: image addition (IA), matrix multiplication (MM), Fibonacci numbers generation (Fib) and sum of absolute differences (SAD).

Kernel	# sim.	Detected (FT scheme) %	Detected (simulator) %	Undetected %	Application crashed %	Escapes % (max. # faults)	Max. # injected faults	Max. # undetected faults	Max. output corruption %	Av. output corruption %
IA	2768	n/a	0	100	0	0	6438	6438	99.66	1.074
MM	621	n/a	0	100	0	43 (9)	50	50	94.75	2.992
Fib	532	n/a	32.33	67.67	0	10.71 (4)	5	5	97.78	53.991
SAD	326	n/a	0	100	0	8.28 (10)	130	130	100	100

3.3.3 Fault Coverage Evaluation

In this section we provide a fault coverage evaluation of ILCOFT-enabled EDDI. The purpose is to determine how ILCOFT affects the fault coverage of EDDI.

We simulate hardware faults by extending the SimpleScalar *sim-outorder* simulator with a fault injection capability. At a specified frequency (every N instructions) a fault is injected by corrupting an input or output register of an instruction (overwriting its content with a random value). Only integer arithmetic instructions are affected by the fault injector. This is because the tested kernels have only integer arithmetic, memory and branch instructions, but the faults inside memory access and branch instructions are not covered by EDDI (only their inputs are protected). Thus ILCOFT-enabled EDDI is also not expected to cover them. Fault injection into an instruction input register simulates faults in hardware structures passing or generating instruction input operands, such as instruction memory, bus or register file. Fault injection into an output register simulates a functional unit fault also. Faults are injected only within the kernel code, because the main function is not protected in our experiments.

We remark that the fault appearance does not represent a realistic model. The aim here is to evaluate the fault coverage of the investigated schemes under different fault pressures (frequencies), and to ensure that as many as possible of the fault propagation paths within the kernels are examined. By making the fault injection periodic rather than random, and by varying the frequency for each of a large number of simulations, we attempt to gain a better control over the process, and to achieve the mentioned goals. Moreover, we simulate burst (multi-bit) faults rather than more probable single-bit faults with the purpose of representing the worst possible case.

Table 3.1, Table 3.2, and Table 3.3 present the faults injection results for the three different schemes. The first column specifies the used kernels. The second column of each table shows the number of simulations executed. The cho-

Table 3.2: Fault injection results for the ILCOFT-enabled EDDI scheme.

Kernel	# sim.	Detected (FT scheme) %	Detected (simulator) %	Undetected %	Application crashed %	Escapes % (max. # faults)	Max. # injected faults	Max. # undetected faults	Max. output corruption %	Av. output corruption %
IA	13526	86.66	0	13.34	0	0.07 (2)	22	11	0.13	0.012
MM	621	53.62	0	46.38	0	23.19 (5)	15	11	99	3.103
Fib	581	66.44	25.99	7.57	0	0	8	3	96.67	38.232
SAD	340	55.88	0	44.12	0	0.29 (1)	25	18	100	100

Table 3.3: Fault injection results for the EDDI scheme.

Kernel	# sim.	Detected (FT scheme) %	Detected (simulator) %	Undetected %	Application crashed %	Escapes % (max. # faults)	Max. # injected faults	Max. # undetected faults	Max. output corruption %	Av. output corruption %
IA	6025	100	0	0	0	0	2	0	0	0
MM	621	100	0	0	0	0	11	0	0	0
Fib	581	67.81	32.01	0.17	0	0	3	2	96.67	96.667
SAD	340	98.53	0	0.29	1.18	0.29 (1)	23	1	100	100

sen number of simulations differs for each kernel, and depends on the number of committed instructions. The frequency of injected faults starts from one fault per 1000 (in some cases 100) instructions, and every new simulation decreases the fault frequency until it becomes roughly one fault per execution. In this way we make sure that all situations from frequent down to rare faults are evaluated, and that random instructions within the kernels are affected. The third column shows how often faults have been detected by the FT scheme. For example, for the IA, 86.66% simulations were aborted with an error message by ILCOFT-enabled EDDI, and 100% by EDDI. The fourth column demonstrates how often errors were detected by the simulator, for example, the application was terminated with an illegal memory access reported. The column labeled "Undetected" contains the percentage of simulations with undetected fault(s), which shows how often the execution finished without reporting errors. The column labeled "Application crashed" demonstrates how often an application crashed, i.e., did not produce any output. The column labeled "Escapes" shows how often escapes occurred, i.e. an application delivered a correct result despite the presence of (undetected) fault(s). The faults have not propagated to the output. In parentheses the maximum number of undetected faults in this situation is given. The column labeled "Max. # injected faults" gives the maximum number of faults injected per execution, before the execution finished either normally or abnormally (was interrupted reporting errors). There are usually fewer injected faults in EDDI than in other schemes, because EDDI detects and reports faults, aborting the execution, earlier. The column labeled "Max. # undetected faults" shows the maximum number of undetected faults, which were injected but not detected; the execution is then finished without reporting errors. Most of the times these undetected faults

result in corrupted application output, except the cases counted in the column labeled “Escapes”. The columns labeled “Max. output corruption” and “Av. output corruption” present the maximum and the average output corruption caused by undetected faults. Only simulations with undetected faults, which finished without reporting errors, are considered here. This demonstrates how many undetected faults propagate to the output, and how much they affect the output. An output corruption percentage is defined as a ratio of the number of wrong output elements generated by an execution to the total number of output elements. The average output corruption is calculated as a sum of all the corruption percentages divided by the number of simulations, i.e., it is the arithmetic average. The average output corruption is used to emphasize that a very high maximum output corruption does not necessarily mean that the output is usually corrupted so much. It can be an exceptional case.

Obviously, ILCOFT affects kernels in very different ways. The difference in the fault coverage can be explained by the density of the duplicated instructions in a kernel. The more instructions are duplicated, the higher the fault coverage is, and the lower the performance and energy consumption gain is. Among the presented kernels, the worst fault coverage (the largest percentage of executions finished with undetected faults) appears in MM and SAD. This is because in these kernels relatively many unprotected computational instructions reside between protected control instructions. Depending on the application, the significant performance increase at the expense of a weaker fault coverage can be considered acceptable. For example, for SAD used in motion estimation, a wrong motion vector leads to a wrong block used for motion estimation, which can usually be tolerated by the user.

The exceptionally high percentage of escapes in MM (with the original and ILCOFT-enabled EDDI schemes) can be explained by the fact that most of the results (output matrix elements) are truncated when overflow occurs. Truncation masks faults by assigning the maximum possible value to any (correct or wrong) larger value. This can also be one of the reasons why MM has a relatively small percentage of detected faults with the ILCOFT-enabled EDDI scheme: the faults are masked before they propagate to a checking instruction which can detect them. With a higher precision (more bits per value), the number of escapes would drop. EDDI does not have any escapes, because it detects all faults in MM.

The most important fault coverage characteristic from the user point of view is the final output corruption. The fact that a certain amount of corruption can be allowed in some applications is the idea behind ILCOFT. Obviously, this

is application-specific and depends solely on the algorithm employed. The IA kernel, computing every pixel value independently, without a long chain of computations, shows very good results for ILCOFT-enabled EDDI: only a few pixels (maximum 0.13% of the whole output image) are corrupted. This is often unnoticed by a user. The maximum output corruption occurred when a fault was injected into the register which held the base address of an array representing one input image line (matrix row), and was later used to fetch all the image data on this line. As a result, garbage was fetched from a random memory location for every pixel of the rest of the line, and the resulting image line was entirely corrupted from the point where the fault appeared. This was quite visible in the output image. It could be solved by performing checks of computed addresses before every load and store, as will be discussed later. Then, only single pixels would have been affected. In all other kernels, besides IA, the resulting values depend on a long chain of computations, and even on each other, so the final output corruption increases dramatically. For example, in Fib, every subsequent value depends on the previous one, and thus, all the values behind the first erroneous one become wrong, independently on the FT scheme used. This leads to the extremely high final output corruption even for EDDI (see Table 3.3). However, only one of 581 simulations (0.17%) finished with undetected errors (2 undetected faults) with EDDI, and 7.57% of the simulations with ILCOFT-enabled EDDI, while 67.67% of unprotected executions finished with undetected errors. The single error undetected by EDDI obviously manifested among the first Fibonacci numbers, so all the following numbers were computed on the base of this error, and thus, about 97% of the final output was corrupted. The average output corruption of about 97% is equal to the maximum, because this is the only undetected error. SAD delivers only one value as a result, which can be either correct or wrong, and any unmasked fault in the computations leads to an error. Consequently, all the undetected and unmasked errors in protected and unprotected executions affect 100% of the output. However, the unprotected execution delivered wrong result in 100% of the simulations, while EDDI-protected did so only in 0.29% of the simulations. The execution protected with ILCOFT-enabled EDDI, as expected, falls in between, delivering wrong output in 44.12% cases.

To investigate the behavior under a more realistic fault appearance model, the same experiments have been conducted with random, rather than periodic, fault injection. Faults into input or output registers were injected at random instructions, with varying fault frequency. The general impression from the results of these experiments is the same as with the periodic fault injection

presented above. However, a few significant differences have been observed, which are discussed below.

For IA protected by ILCOFT-enabled EDDI, the maximum output corruption increased to 21.3%. We explain this by a larger number of faults affecting registers holding array base addresses. For MM, the maximum output corruption decreased to 30% for the non-redundant scheme, and to 50% for ILCOFT-enabled EDDI. With the EDDI scheme, the percentage of detected faults decreased to 73.1%, and the percentage of escapes increased to 26.9%. We attribute these differences to a larger number of faults injected before truncation is performed (the effect of truncation is discussed above). For Fib, the average output corruption decreased to 41.7% for EDDI. This is because fault(s) propagated to the output in more than one simulation (1.4% simulations finished with undetected faults), affecting the output in different ways. For SAD protected by EDDI, the detected percentage dropped to 86.7%. However, most of these faults did not propagate to the output (the percentage of escapes increased to 13%).

The experimental results indicate that the fault coverage of ILCOFT-enabled EDDI can be significantly improved at a relatively low cost. This can be achieved by protecting the computed memory addresses. For example, as mentioned above, it would solve the corrupted output line problem in IA. The protection can be applied before every load and store instruction, by checking the value of the register which holds the memory address. Of course, the redundant value must be computed by a chain of duplicated instructions (which can be done automatically by a compiler). This brings back the trade-off between performance and fault coverage.

The memory address problem is not relevant to EDDI, because the memory is duplicated in this scheme. Thus, all loads and stores reference different memory locations (the original and the duplicates). However, this can be a point where the fault coverage of ILCOFT-enabled EDDI is stronger than that of EDDI itself: EDDI does not have any memory address protection, so a fault in a store instruction can damage any memory location. ILCOFT-enabled EDDI with memory address protection prevents this.

To minimize the performance loss, only the store addresses can be protected, assuming that a memory corruption is more dangerous than fetching a wrong value. But in this case, the IA corrupted line problem discussed above is not solved.

3.4 Application-Level Validation

In this section we discuss how ILCOFT can be applied to an entire application while keeping the programming effort feasible. We estimate the advantages that using ILCOFT brings and evaluate the price to be paid.

As discussed in Section 3.2.2, it is infeasible that the application developer manually annotates all (block) statements with the required FT degree. Instead, all instructions can be automatically protected, and the programmer can focus only on some of the most time consuming application parts to minimize resource consumption at the minimum effort and reliability cost.

The running time of some applications (e.g., from the multimedia domain) is typically dominated by a few kernels or loops. Furthermore, the most time-consuming kernels often feature the natural error tolerance on which the ILCOFT idea is based. Moreover, the most time-consuming kernels are often relatively small, hence it is feasible to manage their protection manually. Thus these kernels favor ILCOFT the most, from the points of view of effectiveness, error tolerance, and minimal programming effort. A significant benefit is expected if the programmer manually protects only (some of) these kernels, while the rest of the application is protected automatically.

We apply this strategy to the *cjpeg* application [142], which compresses an image file to a JPEG file. We have profiled this application and the results show that one of the most time-consuming functions on the simulated architecture is *jpeg_fdct_islow*, which implements the *Inverse Discrete Cosine Transform* (*IDCT*). This function takes from 20.7% to 23.1% of the total execution time, depending on the issue width. However, this function has a relatively small (compared to the whole application) number of static instructions, which makes it easy to manage by hand. We apply ILCOFT-enabled EDDI only to the IDCT kernel (manually), and assume that the rest of the application is protected (automatically) by full EDDI. Further we show how this relatively small programming effort affects the whole application.

The rest of this section is organized as follows. Section 3.4.1 presents and analyzes the performance results, Section 3.4.2 the energy consumption results, and Section 3.4.3 the fault coverage results of this experiment.

3.4.1 Performance Evaluation

We compare the performance of the three schemes using the SimpleScalar simulator tool set [139], as was done in Section 3.3.1. Since currently we do

not have an automatic tool implementing EDDI protection, we only apply the FT schemes to the IDCT kernel in the simulation. We measure the application execution time with the non-redundant IDCT kernel, with EDDI and ILCOFT-enabled EDDI protection. Then we use these results to derive expected results for the completely protected application. Specifically, let the total running time of cjpeg be given by

$$T_{total} = T_{idct} + T_{rest},$$

where T_{idct} is the time taken by the IDCT kernel and T_{rest} is the time of the rest of the application. Furthermore, assume that applying full EDDI (using a tool) to the rest of the application slows it down by a factor of f, then the total running time of cjpeg protected with EDDI is given by

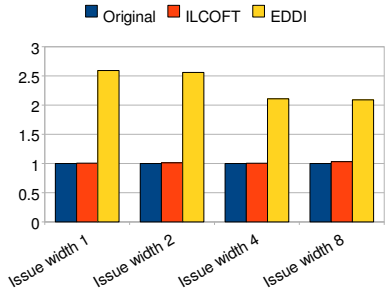
$$T_{total\text{-}eddi} = T_{idct\text{-}eddi} + f \cdot T_{rest},$$

where $T_{idct\text{-}eddi}$ is the measured running time of the IDCT kernel when protected with EDDI. Similarly, the total running time of cjpeg protected with ILCOFT-enabled EDDI is

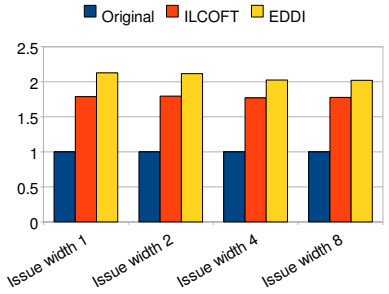
$$T_{total\text{-}ilcoft} = T_{idct\text{-}ilcoft} + f \cdot T_{rest},$$

where $T_{idct\text{-}ilcoft}$ is the measured running time of the IDCT kernel when protected with ILCOFT-enabled EDDI. In other words, for the IDCT kernel we take the measured running time and for the rest which is protected by full EDDI we assume an overhead by the factor of f (which is the same for both $T_{total\text{-}eddi}$ and $T_{total\text{-}ilcoft}$). We assume that EDDI incurs an overhead of 100%, which is quite pessimistic for higher issue widths, because EDDI increases the amount of ILP, as was shown in Section 3.3.1. Note that the less overhead EDDI introduces in the rest of the application, the more pronounced benefits ILCOFT-enabled EDDI attains. 100% EDDI overhead means that the factor f equals 2 in our estimations.

Figure 3.6(a) presents the slowdown of the IDCT kernel protected with EDDI and ILCOFT-enabled EDDI over its non-redundant version. The runtime of all the IDCT function invocations during JPEG encoding is accumulated. This figure reflects the simulation results. Unlike in Section 3.3.1, this experiment uses a slightly modified version of EDDI, which does not duplicate memory. Duplicating (allocating and copying) all the used memory for each invocation of the IDCT function would incur a significant unjustified overhead in the application. This overhead would not reflect the actual EDDI influence, because in full EDDI some parts of the used memory would be duplicated at other places, possibly only once instead of on every IDCT function invocation. One



(a) Simulated IDCT kernel slowdown, accumulated for all its invocations in JPEG encoding.



(b) Estimated JPEG encoder slowdown with protection applied to the whole application. Based on the assumption that for the rest of the application (except the IDCT kernel), EDDI introduces 100% overhead.

Figure 3.6: Slowdown of EDDI and ILCOFT-enabled EDDI versions over the non-redundant version, for varying issue widths.

would expect EDDI to incur a lower overhead than depicted in Figure 3.6(a), because the duplicated instructions are independent of the original ones, so the available ILP is increased. However, for the IDCT kernel EDDI introduces an overhead of larger than 2, for example, 2.59 for issue width 1 and 2.11 for issue width 4 (see Figure 3.6(a)). This is due to the high register utilization in IDCT. EDDI halves the number of available integer registers, allocating 13 (out of 32) of them, leading to the need for a large amount of register spilling. Besides the additional memory overhead from register spilling, the stored value of every store instruction should be checked in EDDI, which significantly increases the number of inserted branch instructions. The poor scaling with the increased issue width is due to the fact that the original IDCT code has sufficient independent instructions that can be executed in parallel, so the additional ILP introduced by EDDI cannot be fully utilized due to the lack of computing resources.

Unlike with EDDI, IDCT is very friendly to ILCOFT (when only the control instructions are considered critical). There are only two loops in the *jpeg_fdct_islow* function, which are not nested. Therefore, ILCOFT-enabled EDDI allocates only one register (for shadow copies of the counters), duplicates only a few instructions, and adds only two checks. This redundancy is very small for a function with about 350 static instructions, which leads to a negligible performance overhead over the original.

Table 3.4: Fault injection results for the JPEG encoding.

	# sim.	Detected (FT scheme) %	Detected (application) %	Detected (simulator) %	Undetected %	Application crashed %	Escapes % (max. # faults)	Max. # injected faults	Max. # undetected faults
ILCOFT	100	0	59	4	35	0	9(4)	9	8
EDDI	100	98	0	27	0	0	0	147	0

Figure 3.6(b) presents the analytically estimated performance overhead for the whole JPEG encoder. Here the IDCT kernel is protected with either EDDI or ILCOFT-enabled EDDI, and the rest of the application with EDDI. It shows that applying ILCOFT to only the IDCT kernel of EDDI-protected JPEG encoder is able to deliver a performance gain of 14% on average.

3.4.2 Energy Consumption

Similar to Section 3.3.2, we obtained the energy consumption results with Wattch [141]. The results show a behavior similar to that of the performance in Figure 3.6. On average, the JPEG encoder with the IDCT kernel protected by ILCOFT-enabled EDDI consumes about 14% less energy than with EDDI.

3.4.3 Fault Coverage Evaluation

Finally, we performed experiments similar to those in Section 3.3.3, injecting faults regularly (from once per 500 thousand to once per 50 million instructions) into the input and output registers of the instructions within the IDCT kernel. The fault frequency decreases with each simulation. Table 3.4 presents the fault injection results for 100 simulations. The presentation is similar to that of Table 3.1.

Table 3.4 shows that EDDI detected almost all faults (98%). The effects of the other 2% faults have been masked, did not propagate to the output, or were detected by the simulator before the FT scheme. ILCOFT-enabled EDDI, on the other hand, did not detect any faults in our simulation. We explain this by the fact that the number of checks performed within the kernel is very small compared to EDDI.

The column “Detected (application)” shows that 59% of the faults in ILCOFT-enabled EDDI have been detected by the application, reporting an out-of-range DCT coefficient. Nothing has been detected by the application in EDDI, because the faults have been caught before by EDDI or the operating system. The column “Detected (simulator)” demonstrates that for both schemes some

faults have been detected by the simulator (operating system), reporting, for example, an illegal memory access. From the column "Application crashed" it can be seen that the application never crashed due to the faults, neither for EDDI nor for ILCOFT-enabled EDDI.

It may appear surprising that in the column "Max. # injected faults", the maximum number of faults injected per simulation is much larger for EDDI than for ILCOFT-enabled EDDI. This is due to the way our error handler works: when EDDI detects a fault, it reports an error and returns from the running IDCT function, but does not stop the whole application. In this way we are able to see if EDDI detects faults in future IDCT invocations. However, unlike in EDDI, undetected faults in ILCOFT-enabled EDDI easily propagate to the points where the wrong values are used in loads and stores, which triggers a simulator (operating system) exception, and the application stops. Thus, the simulation is shorter, and the fault injector is not able to inject more faults.

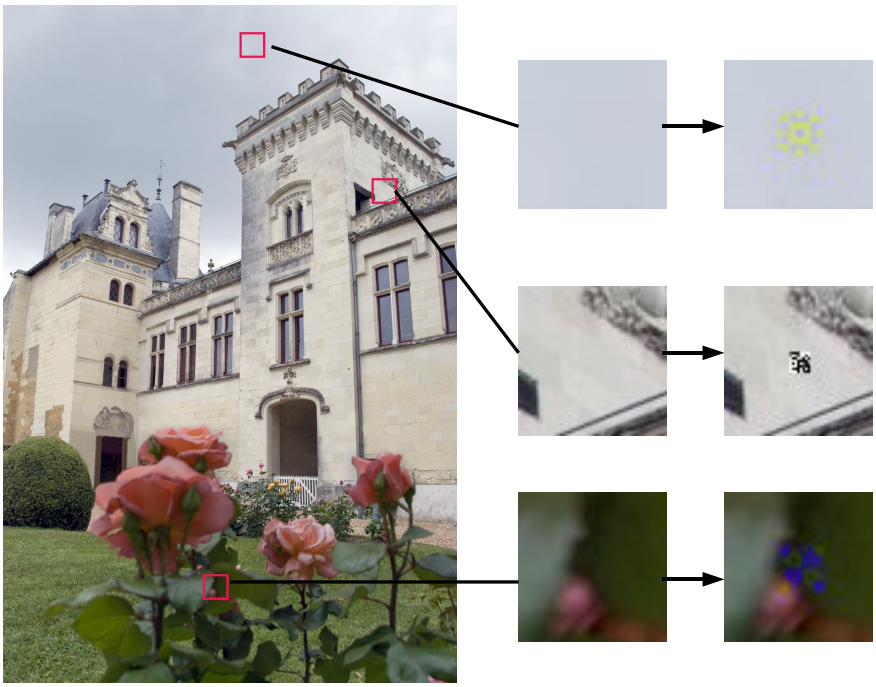


Figure 3.7: Output corruptions due to the undetected faults in IDCT.

35% of the ILCOFT scheme simulations ended with undetected faults, which were either masked or propagated to the output. Figure 3.7 depicts one of the most corrupted output JPEG images produced by our simulations. It was pro-

duced with a fault frequency of once per 10 million instructions, and this particular simulation produced the maximum number of undetected faults (equal to 8). The three areas where we were able to visually recognize corruption are marked with squares. On the right, the magnified versions of these areas are shown, in the original image (left column) and the corrupted image (right column).

In our opinion such output corruptions are quite an acceptable price for the significant speedup and energy saving which ILCOFT provides. This is under the assumption of relatively low requirements to the output image quality, which can be quite appropriate in embedded systems, PCs, and other systems not designed for critical missions. For applications with very high requirements to the output image quality JPEG is not a good choice anyway, because it is intrinsically lossy. Moreover, relatively low fault rates are anticipated in the foreseeable future in normal environments (ILCOFT does not target extreme cases such as environments with high radiation). The low fault rates mean that most of the time redundancy is not useful, but only incurs overhead. In this situation, reducing the time and energy overhead, still being guaranteed against severe crashes, but increasing the chance of tolerable errors, is a valid option.

3.5 Conclusions

In this chapter we have proposed instruction-level, rather than application-level, configurability of FT techniques. This idea is based on the observation that some applications pose different FT requirements on their various parts. For example, in multimedia applications, an error in parts calculating the value of a pixel, a motion vector, or a sample frequency (sound) is usually unnoticed or ignored by a human observer. An error in the control (critical) part, however, will probably lead to a crash of the whole application. This indicates that it is most important to apply the strongest FT features to the critical parts, while non-critical parts can be protected with a weaker FT (or left unprotected) to improve the application performance and reduce energy consumption. In applications with execution time constraints (e.g. real-time applications), the time saved by reducing the FT of non-critical parts can be used to further increase the FT of the critical parts, thus improving the overall application reliability.

We have shown how several existing FT schemes could be adapted to support ILCOFT. We also proposed a way for a programmer to specify the desired degree of FT in a high-level language or assembly code, and indicated how a

compiler could apply FT techniques to control code automatically.

The experimental results have demonstrated that ILCOFT is able to significantly improve an application performance and reduce the energy consumption when applying a higher FT degree to its critical instructions only. At the kernel level, the performance and energy dissipation improved by up to 50%, and at the application level by up to 16%. In the application-level experiments, improvements are achieved by applying ILCOFT to only one of the most time-consuming kernels, thereby reducing the programmer effort. The results show that adaptation of only one kernel provides a significant application-level improvement.

The price to be paid for the performance and energy gains provided by ILCOFT is the smaller fault coverage. The experimental results have also shown that the fault coverage of ILCOFT-enabled EDDI is very application-specific and is comparable to the fault coverage of full EDDI for applications that compute independent elements. The fault coverage certainly depends on the amount of redundancy applied. In some cases the output corruption allowed by ILCOFT is tolerable, in others it is not. Finally, we have demonstrated that adding memory address protection in ILCOFT-enabled EDDI could significantly improve the fault coverage.

Note. This chapter is based on the following papers:

Demid Borodin, B.H.H. (Ben) Juurlink, and Stamatis Vassiliadis, **Instruction-Level Fault Tolerance Configurability**, *IC-SAMOS VII: International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation*, pp. 110–117, July 2007.

Demid Borodin, B.H.H. (Ben) Juurlink, Said Hamdioui, and Stamatis Vassiliadis, **Instruction-Level Fault Tolerance Configurability**, *Journal of Signal Processing Systems*, Volume 57, Issue 1, pp. 89–105, October 2009.

4

Instruction Vulnerability Factor

The previous chapter has presented the ILCOFT approach, which enables a developer to assign different protection levels to various application parts. The programmer can manually assign the required degree of FT to instructions or blocks of instructions. This, however, requires significant effort, and is very error-prone. Alternatively, a compiler can assign the degree of FT automatically, assuming that only control-flow instructions are critical. This approach is save only for a limited set of applications, because faults in data processing instructions can cause completely wrong output in many applications. Other automatic methods to assign the required degree of FT are desirable, able to evaluate how critical every instruction is for the final application output.

This chapter introduces the notion of *Instruction Vulnerability Factor* (**IVF**), which determines how much of the final application output is corrupted due to fault(s) in every instruction. ILCOFT uses IVF to determine the necessary degree of FT for executed instructions.

This chapter is organized as follows. Section 4.1 gives an introduction and motivation. Section 4.2 describes the IVF estimation performed in this work, and how it is then used to reduce the overhead of the instruction duplication error detection technique. Section 4.3 presents the experimental results evaluating the performance penalty reduction and fault coverage of the proposed IVF-based ILCOFT technique. Finally, Section 4.4 draws conclusions and describes future work.

4.1 Introduction

The ILCOFT scheme as presented in Chapter 3 and [80, 94] requires the application developer to manually assign the required protection level to instructions. To process (a large part of) a relatively large application, significant effort is needed. In addition, this process is error-prone because it entirely relies on the programmer's judgment of every instruction's vulnerability. An alternative approach is if the compiler automatically assigns the required protection levels. Because the compiler cannot evaluate the criticality of data processing instructions, however, this method is feasible if only control flow instructions need strong protection. Thus, all control flow instructions (branches, jumps, function calls, etc.) and instructions on which they depend (address calculations, condition evaluations, etc.) are assigned a high protection level. Sundaram et al. [143] discuss such compiler analysis in detail. This automatic compiler-based method, however, is based on the assumption that no data manipulating instructions are critical, which is not safe. In many applications faults in data processing instructions can corrupt the whole application output, in which case, even though the application does not crash, it is not usable. This approach can only be safely applied to a very limited set of applications. A more sophisticated method to assess every instruction's criticality is required for most applications.

To address this instruction criticality assessment problem, this chapter introduces the notion of *Instruction Vulnerability Factor* (*IVF*). IVF is analogous to the Architecture Vulnerability Factor (AVF) [144], but addresses application instructions instead of hardware structures. AVF estimates the probability that a fault in a particular hardware structure will result in an error visible in the final application output. Experiments demonstrate that different processor structures have different AVFs [144]. For example, unlike faults in ALUs, faults in branch predictors can only result in a reduced performance, but cannot damage the final application output. We propose that a similar metric is applied to the application's instructions. Faulty results of different instructions affect the final application output in different ways. IVF measures how much of the final output is corrupted due to faults in every instruction.

IVF can be used to determine the appropriate protection level that ILCOFT assigns to instructions. Instructions with a higher IVF should be protected better, while others can be assigned a weaker protection level to reduce the overhead.

To compute the IVF for every instruction, off-line profiling is performed. It

can be done either in a simulation environment or in real hardware capable of injecting faults. Fault(s) should be injected into every instruction, one per experiment, and the resulting application output be used to estimate the instruction IVF. Multiple experiments per instruction are preferable to obtain statistically valid results. To reduce the amount of work this process requires, only the most time consuming application parts can be processed. As will be shown in Section 4.3, this is able to provide significant application-level advantages. This procedure has to be performed off-line, only once per application. The result (IVF value for all the considered instructions) is then stored and distributed together with the application binary code. At runtime, every instruction is protected at the level required by its IVF value.

4.2 IVF and IVF-Based ILCOFT

This section introduces the IVF and demonstrates how it can be used by ILCOFT to assign the required protection levels to different instructions. Section 4.2.1 discusses how the IVF can be estimated. Section 4.2.2 presents a time redundant error detection scheme which duplicates instructions in the pipeline and compares results produced by replicas. To decrease the performance penalty due to error detection, this instruction duplication scheme is adapted to support IVF-based ILCOFT in Section 4.2.3.

4.2.1 IVF Estimation

IVF estimation requires monitoring how different faults in every instruction propagate to the final application output. In other words, how much of the output is corrupted due to faults in every instruction. This can be achieved using fault injection experiments, either in a simulation environment or on fault injection-enabled hardware. Alternatively, this can also be done in software. To simulate hardware faults in software, the correct machine instructions can be substituted with other instructions producing wrong results in the same output registers. This software solution can be expected to significantly speed up the IVF estimation process, but it reduces the fault injection flexibility. Faults cannot be injected into hardware structures, exploring all the effects they might have. Instead, only the visible errors triggered by these faults are simulated.

Ideally, all possible faults have to be injected one at a time into every executed instruction, and the corresponding output corruption measured. Moreover, some faults in dependent instructions can affect each other, thus combi-

nations of faults in different instructions should also ideally be examined. This, however, would require an enormous number of experiments which is not feasible. Our Experimental results (see Section 4.3.4) demonstrate, however, that injecting several random faults into every instruction provides a sufficiently accurate IVF estimation. It is desirable to perform multiple experiments with every instruction to achieve statistically valid results, because different faults in the same instruction can affect the execution in different ways.

In this work, the IVF estimation is performed using the *sim-outorder* simulator from the SimpleScalar tool set [140]. During every simulation, the output of one instruction execution is assigned a random value, simulating the worst case (a multi-bit) transient fault. Faults are only injected into instructions producing results, such as arithmetic operations and memory loads. Memory stores and jumps are not affected, because the error detection scheme used in this work does not cover faults in these instructions. The final application output is then compared to the correct one, and the output corruption is measured. Depending on the application, this can be the percentage of corrupted bytes in the output (in an image for instance), or the percentage of corrupted output items, such as matrix elements. The output corruption percentage is then saved as the instruction's IVF. When multiple experiments per instruction are performed, the average IVF is saved for all the experiments.

The average IVF value is computed for every static instruction within the considered code segment. This requires a large number of simulations, especially if multiple experiments per instruction are conducted (as is desirable). This, however, has to be done only once per application, and can be done off-line. The collected statistics are then saved together with the application binary code and used at runtime. For large applications, it is not even necessary to compute the IVF of every instruction. It can be estimated only for the instructions within the most time consuming application parts, such as multimedia kernels. Section 4.3.3 will demonstrate that this approach achieves significant application-level performance improvements.

The IVF information can be saved in a program binary code as a separate table. This table maps application instructions (identified by their PC) to the corresponding IVF values, or to the required protection levels. Storing the protection levels is likely to take less space (for example, only one bit is needed if two protection levels are available), and no logic is needed to determine the protection level from the IVF value. On the other hand, storing the IVF value provides more flexibility, since the system can be configured at runtime to define the threshold IVF values controlling the required protection levels. The

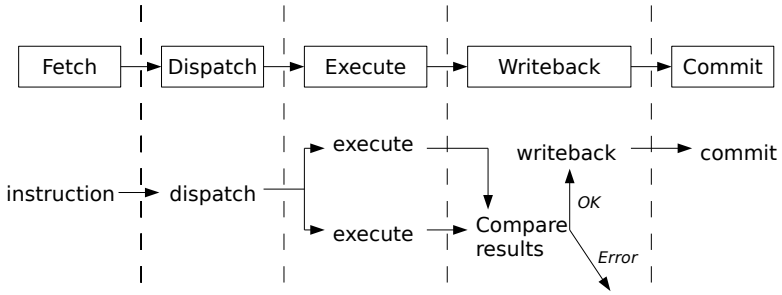


Figure 4.1: Instruction duplication.

IVF information stored in a separate table needs to be loaded into a special hardware buffer (possibly simultaneously assigning the protection levels and storing them instead of IVF values to reduce the storage requirements). Then, the table must be looked up to determine the proper protection level for every executed instruction. The table can be compressed. For example, with only two protection levels available in the system (one of which is the default), only PCs of instructions with the non-default required protection level have to be stored. Alternatively, instead of keeping the IVF data in a separate table, it might be preferable to include a field defining the required protection level in the instruction format.

4.2.2 Instruction Duplication

In this chapter a hardware ILCOFT scheme is used. Instruction duplication in the pipeline is used as a time-redundant error detection technique, and its overhead is minimized using the IVF information collected as described in Section 4.2.1. The instruction duplication scheme is based on [69] (see Section 2.4.2), but executes every instruction twice rather than duplicating them in the dynamic scheduler. This requires less space in the dynamic scheduler, but does not protect against faults in it. Figure 4.1 depicts the instruction execution steps with the corresponding activities performed by every instruction. A fetched instruction proceeds normally until the execution stage. There the instruction is kept in the RUU until it has been issued twice. When the results of both executions are available, they are compared in the Writeback stage, and the instruction commits if no errors are detected.

What happens if the results do not match depends on the goals of the system.

A system targeting fail-safe operation would signal an error and halt. If FT is required, another copy of the questioned instruction could be created and executed, performing a majority voting on all the obtained results according to the TMR scheme (see Section 2.3), or a different form of recovery could be initiated. This, however, is outside the scope of this work.

All instructions except memory stores are duplicated. Stores consist of two operations: effective address calculation and the store itself. The address calculation is duplicated and verified. However, it does not make sense to duplicate the store operation itself, because the result cannot be verified.

In the considered implementation, the choice of the FUs used to execute the redundant instruction copies is only driven by resource availability. Even if multiple appropriate FUs are present, it can happen that the redundant copies still execute on the same FU. This means that permanent, long-lasting transient, and common faults in the FUs and data buses are not always covered, since the redundant copies might be affected in the same way and produce the same (wrong) results. In Chapter 5 we present an approach that improves the long-lasting transient and permanent fault coverage of this scheme.

4.2.3 IVF-Based Selective Instruction Duplication

To support ILCOFT, the FT technique(s) used in the system have to be able to apply different protection levels to different instructions. The higher the IVF is (and thus the more critical the instruction is), the stronger the protection applied to the instruction should be. In the simplest case, only two protection levels are available: an instruction is either protected or left unprotected. This case is very practical, because more complex protection schemes with multiple protection levels can be very expensive. In this work such simple protection scheme based on instruction duplication is used, called *IVF-Based Selective Instruction Duplication* (*IVF-SID*). Depending on its IVF, an executed instruction is either duplicated and the result is verified (as described in Section 4.2.2), or it is executed only once without error detection.

Note that two terms are used in this chapter: IVF-based ILCOFT and IVF-SID. IVF-based ILCOFT is a general ILCOFT technique which decides how critical instructions are based on their IVF values. IVF-based ILCOFT can be applied to many different FT techniques able to protect individual instructions. IVF-based ILCOFT applied to the instruction duplication error detection technique is called IVF-SID.

The simple protected/unprotected scheme requires to establish a threshold IVF

value, further referred to as IVF_{thr}. Instructions with IVF above IVF_{thr} should be protected, and instructions with IVF below IVF_{thr} are left unprotected. The IVF_{thr} value should be carefully chosen to guarantee that the amount of application output corruption it represents is indeed tolerable. Ideally it should be determined specifically for every application, because different applications have varying output damage tolerance.

Figure 4.2 shows a histogram of IVF values for different kernels and applications (the benchmarks are described in Section 4.3.1). The horizontal axis represents different IVF values from 0% to 100%. The vertical axis shows the percentage of static instructions in different benchmarks that have the corresponding IVF value. Figure 4.2 clearly illustrates that the majority of IVF values are at the extremes, they are either larger than 99% or less than 1%. Some kernels (Image Addition and SAD) do not have any instructions with IVF values in between. Most instructions either corrupt (almost) the whole output or less than 1% of it. This indicates that 1% is a good IVF_{thr} value. Damage of 1% of the output can be considered tolerable for many application domains, including many multimedia applications. Increasing the IVF_{thr} to, for example, 10% would not lead to significant changes, because most benchmarks do not have any instructions with an IVF in the range of 1% to 10% (see Figure 4.2). Thus, an IVF_{thr} value of 1% is used for IVF-SID in this work. Instructions whose wrong results corrupt less than 1% of the application output are not protected.

The IVF values distribution shown in Figure 4.2 suggests that a simple protected/unprotected ILCOFT scheme is sufficient. Multiple available protection levels would not be very useful for most of the benchmarks shown in Figure 4.2, because they have only a few instructions with medium IVF values. On average 44% of the instructions have an IVF below 1%, and 39% of the instructions have an IVF above 99%. Most of the other instructions have an IVF below 5% or above 95%. Only the ADPCM encoder and decoder have a significant number of instructions (10% or more) with an IVF in the range 40-90%. For such applications, the following scheme could be used: no or minimum protection for instructions with an IVF below 1%, average protection for instructions with an IVF between 1% and 99%, and maximum protection for instructions with an IVF above 99%.

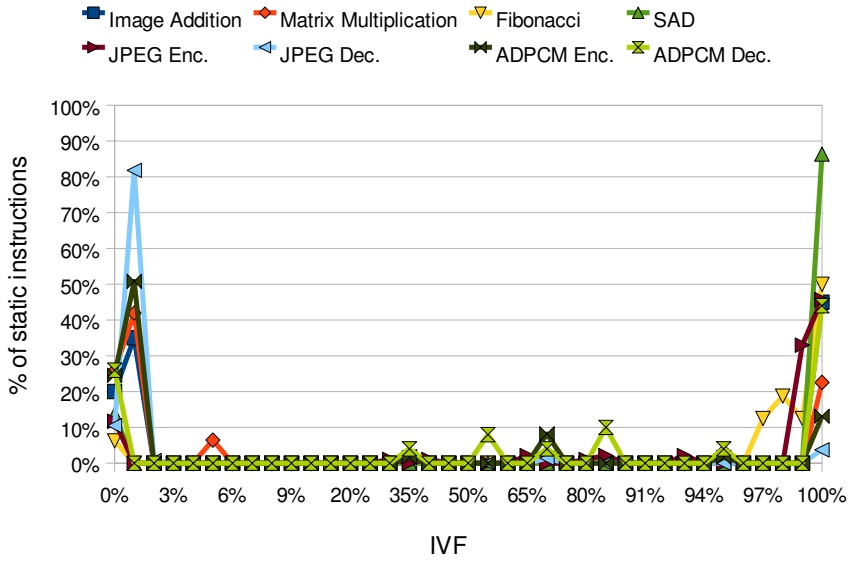


Figure 4.2: Histogram of IVF values for different applications and kernels.

4.3 Experimental Evaluation

This section experimentally evaluates the proposed IVF-SID method. Section 4.3.1 describes the used simulator and benchmarks. Section 4.3.2 explains the details of the employed IVF estimation method. Section 4.3.3 demonstrates the performance improvements IVF-SID achieves compared to duplicating all instructions. Section 4.3.4 evaluates its fault coverage. Finally, Section 4.3.5 compares IVF-SID with a manual ILCOFT method used in Chapter 3.

4.3.1 Experimental Setup

As in Chapter 3, simulations are performed using the *sim-outorder* simulator from the SimpleScalar tool set. Section 4.2.1 describes how the simulator was used to estimate IVF for different benchmark instructions. The machine configuration, chosen to represent a moderately powerful processor suitable both for general purpose and embedded computing, is shown in Table 4.1.

Four kernels (same as in Chapter 3, see Section 3.3) and four full applications are used as benchmarks. Encoders and decoders for JPEG image compression and ADPCM sound compression are the full applications used, taken from the

Table 4.1: Processor configuration.

Fetch/Dec./Issue Width	2, 4, or 8
# of Int. ALUs	2, 4, or 8
# of Int. Mult./Div.	1
# of FP ALUs	1
# of FP Mult./Div.	1
RUU Size	64
Memory Latency	112 cycles (first chunk), 2 cycles (subsequent chunks)
L1 Data Cache	32 KB, 2-way set associative
L1 Instruction Cache	32 KB, 2-way set associative
L2 Unified Cache	512 KB, 4-way set associative

MediaBench benchmark suite [145].

4.3.2 IVF Calculation

The IVF is estimated as described in Section 4.2.1. For the full applications, to reduce the simulation time and demonstrate that it still provides useful results, IVF-based ILCOFT is applied only to the most time consuming functions. As in Chapter 3, the applications are profiled and the most time-consuming functions identified (for example, the *forward_DCT* function in the JPEG encoder, and the *jpeg_idct_islow* function in the JPEG decoder). The IVF values are estimated and are later used by ILCOFT only for the instructions within these functions. In other parts of the applications all instructions have been duplicated as described in Section 4.2.2.

To calculate the IVF of every instruction, ten fault injection experiments per static instruction have been conducted for the kernels, and three experiments per instruction for full applications. To evaluate the accuracy of this method, an additional experiment is performed with one of the kernels (Matrix Multiplication). This experiment attempts to approach the ideal (exhaustive) case taking into account all possible faults and even combinations of faults in different instructions (see Section 4.2.1). First, for every evaluated static instruction, one thousand experiments injecting a single random fault into it are executed. Then, faults are injected into two different instructions per experiment. This is achieved in the following way. Each instruction is coupled with every other instruction in the kernel whose originally estimated IVF (from the first experiment with ten injections per instruction) does not exceed its own originally

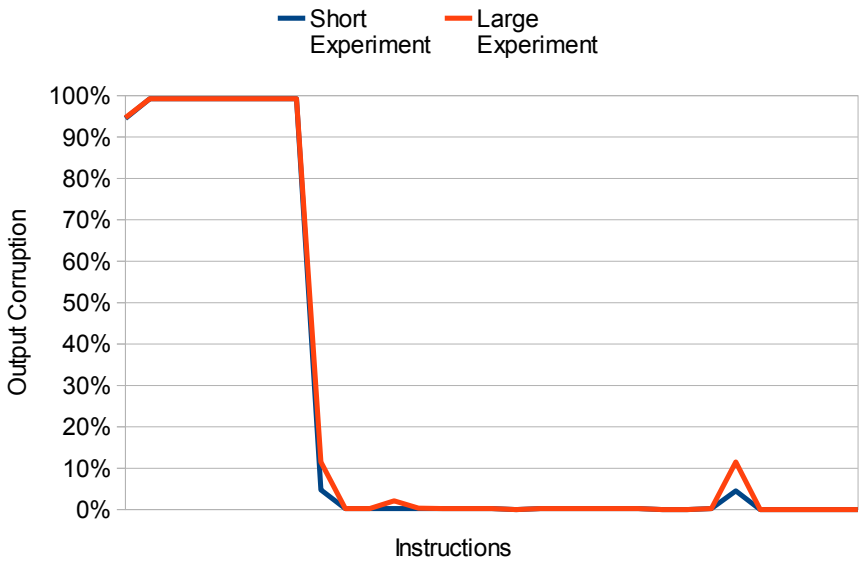


Figure 4.3: Results obtained with the short and large Matrix Multiplication IVF estimation experiments.

estimated IVF. The obtained output corruption contributes to the IVF of the evaluated instruction. Instructions whose originally estimated IVF is larger than that of the evaluated instruction are not coupled, because they affect the output more than the evaluated instruction. Fifty iterations of this double-injection experiment are performed, resulting in 350 to 1500 simulations per instruction (depending on the number of coupled instructions).

Figure 4.3 compares the results of the short and large Matrix Multiplication IVF estimation experiments. The horizontal axis represents different application static instructions. The vertical axis shows the output corruption percentage due to faults injected into the corresponding instruction. Figure 4.3 shows that the results of the short and large experiments closely follow each other. The average difference between the obtained IVF values is 0.5%. The maximum difference is 7% (it is so large only for two instructions). Hence, the short experiment is sufficiently accurate. The fault injection experiments described in Section 4.3.4 confirm that a high level of reliability is achieved using the short IVF estimation experiments.

To provide an insight on the time the IVF estimation process takes, the fault injection experiment simulation time has been measured for one randomly cho-

sen kernel and one application (on a Pentium 4 machine with 3 GB RAM). For the Fibonacci kernel, an experiment injecting one fault into one instruction took 0.8 seconds, and for the JPEG Encoding application it took 136.7 seconds. This is for the case when simulation produces a complete output (does not crash due to faults). Simulations crashing (or producing incomplete outputs) due to faults take less time. On the other hand, there are also faults substantially increasing the simulation time. For example, a fault that significantly increases the value of a variable controlling the exit condition of a large loop can have such effect.

4.3.3 Performance Evaluation

Instruction duplication requires that every dynamic instruction is executed twice. Since duplicated instructions are independent, they can be executed in parallel, provided that sufficient computational resources are available. Thus, instruction duplication increases the amount of ILP available in the application. Unless the original application has a very limited amount of ILP, instruction duplication is likely to introduce a significant performance penalty due to the lack of computational resources available to execute the instruction duplicates. By avoiding the re-execution of instructions with an IVF smaller than 1%, IVF-SID reduces the overhead of instruction duplication.

Figure 4.4 shows the performance overhead of full instruction duplication and IVF-SID over the original execution without any protective redundancy. The benchmarks are executed on a system with four integer ALUs, and a fetch/decode/issue width of four. Figure 4.4 demonstrates that for all the benchmarks, IVF-SID recovers a certain amount of the performance overhead due to instruction duplication. For example, if full duplication is 80% slower than the redundancy free execution, and IVF-SID is 40% slower, than the amount of performance it recovers is 50%. Figure 4.5 shows the amount of recovered performance for different benchmarks. Three machine configurations are considered: with two, four and eight integer ALUs. To balance the machine organization, the fetch, decode, and issue width matches the number of integer ALUs (see Table 4.1).

For the kernels, the recovered performance varies from 1.1% (for Fibonacci, on a system with eight integer ALUs) to 77.7% (for Matrix Multiplication, on a system with two integer ALUs). For full applications the recovered performance varies from 1% (JPEG Encoder, system with two integer ALUs) to 58.3% (ADPCM Encoder, system with two integer ALUs). For full applications, the whole execution time is measured, including the I/O overhead.

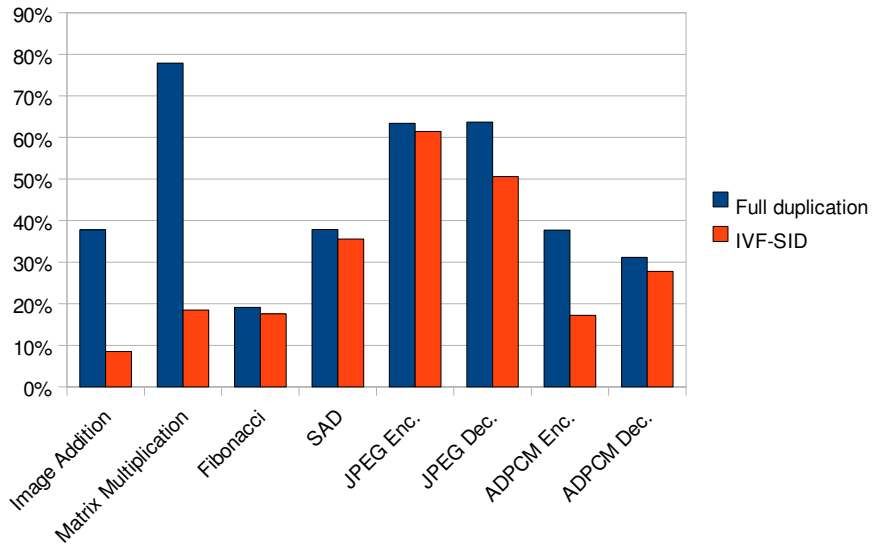


Figure 4.4: Performance penalty of instruction duplication and IVF-SID compared to redundancy-free execution. System with 4 integer ALUs, and a fetch/decode/issue width of 4.

The amount of recovered performance depends on the IVF distribution of the application, the amount of ILP available in the redundancy-free application, and the processor issue/fetch/decode width. The more instructions with low IVF are found in the application, the fewer executed instructions are duplicated, and more performance is recovered. The IVF distribution of the benchmarks is shown in Figure 4.2.

In SAD, almost 86.4% of the static instructions have an IVF of above 99% (and thus definitely need to be duplicated in IVF-SID), and only 13.6% of the instructions have an IVF lower than 1% (and thus do not need to be duplicated). This is because SAD produces only one output, which can be either correct (100% of the output is correct) or wrong (100% of the output is damaged). Any fault which propagates to the final SAD output damages 100% of it. Therefore, only faults that do not propagate to the final output at all (damage unused data, or faults that are masked) can be tolerated in SAD. We call such faults *escapes*. Due to these characteristics, SAD is one of the worst performing benchmarks: only 3.4% to 10.8% of performance is recovered in IVF-SID, depending on the processor issue/fetch/decode width.

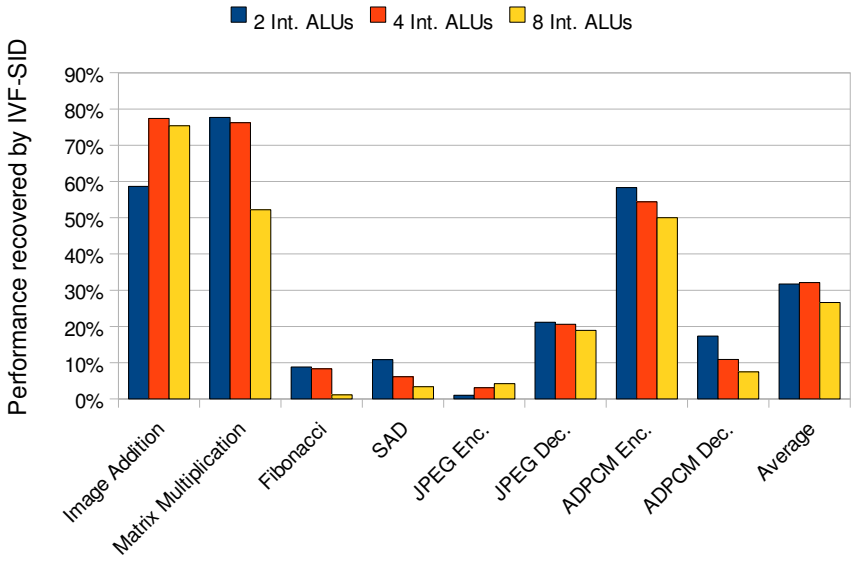


Figure 4.5: Performance penalty reduction achieved by IVF-SID over instruction duplication on different processor configurations. Fetch/decode/issue width matches the number of integer ALUs.

A similar situation appears with the Fibonacci numbers generator, for which IVF-SID recovers from 1.1% to 8.8%. The generator produces a series of numbers, every one of which depends on the previous values. Thus, an error appearing in the beginning of the sequence damages all the subsequent numbers. Only 6.3% of the static instructions in the Fibonacci numbers generator have an IVF of below 1%. 50% of the static instructions have an IVF of 99% or 100%, and 43.8% have an IVF in the range 95%-99% (see Figure 4.2).

The highest performance improvements achieved by IVF-SID are obtained for Image Addition (58.7% to 77.4%) and Matrix Multiplication (52.2% to 77.7%). In Image Addition, 55% of the static instructions have an IVF of below 1%, and 45% have an IVF of above 99%. In Matrix Multiplication, 67.7% of the static instructions have an IVF of below 1%, and 22.6% have an IVF of above 99%. Both these benchmarks produce many independent output elements. When one of them is damaged, it often corrupts less than 1% of the whole output, which is the used IVF_{thr} value. For large images with millions of pixels which are common nowadays, one wrong pixel will most probably not even be visible.

Figure 4.5 shows that IVF-SID improves the performance of the ADPCM encoder much more than the other full applications. This is due to the fact that the encoding function in the ADPCM encoder, to which IVF-based ILCOFT is applied, is a substantial part of the whole application. In contrast, the IVF-enabled *forward_DCT* function in the JPEG encoder is only a part of the whole application, which also calls many other functions. Thus, the ADPCM encoder (and also decoder) functions affect the whole application performance much more than the functions in the JPEG encoder and decoder to which IVF-based ILCOFT is applied. The ADPCM encoder, however, has significantly more instructions with an IVF below IVF_{thr} (75.4% of the instructions, compared to 26% in the decoder), and thus it achieves a much higher performance improvement.

Figure 4.5 also shows that in most cases, the performance recovered by IVF-SID drops when the number of integer ALUs (and the fetch/decode/issue width) increases. This is due to the larger amount of computational resources available. The ILP introduced by instruction duplication is utilized more effectively on systems with more integer ALUs. Thus, the reduction of the number of executed instructions due to IVF-SID does not affect the performance so much. In contrast, for Image Addition and the JPEG Encoder, IVF-SID recovers less performance on a system with two integer ALUs than with four of them. We attribute this to the issue width, which is insufficient and becomes a bottleneck in these cases. Increasing the fetch/decode/issue width from two to four (while keeping two integer ALUs) increases the recovered performance from 58.7% to 70.3% for Image Addition.

The IVF values distribution in static application instructions cannot be expected to always perfectly indicate how useful the IVF-based ILCOFT will be for the program. This is because performance depends on the dynamic behavior, on how many times every static instruction with a certain IVF will be executed.

4.3.4 Fault Coverage

To evaluate the fault coverage of IVF-SID with IVF_{thr} equal 1%, multiple fault injection experiments have been conducted. Every benchmark has been executed around 100 times, with one fault in the output of a random instruction within the kernel to which IVF-SID is applied. To simulate the worst-case scenario, burst (multi-bit) faults are injected rather than single-bit ones. This

Table 4.2: Fault coverage statistics.

Benchmark	Detected (%)	Wrong output (%)	Escapes (%)	Max. output damage (%)	Min. output damage (%)	Average output damage (%)
Image Addition	43.16	48.42	8.42	0.06	0	0.05
Matrix Multiplication	59.14	18.28	22.58	0.25	0.25	0.25
Fibonacci	95.92	0	4.08	0	0	0
SAD	95	0	5	0	0	0
JPEG Encoding	76.77	0	23.23	0	0	0
JPEG Decoding	10	78	12	0.03	0	0.01
ADPCM Encoding	33.33	31.31	35.35	0.64	0	0.12
ADPCM Decoding	87.76	0	12.24	0	0	0

is achieved by changing the instruction output to a random value. There are two reasons to inject only one fault per simulation. First, the current and near-future fault rates are not very high, and thus, more than one fault in a relatively small kernel is unlikely. Second, when injecting multiple faults per experiment, the chance that at least one of them will be detected increases. As a result, other faults that would not have been detected alone are hidden, and their effect on the output is not investigated.

Table 4.2 depicts the collected statistics. The second column shows the percentage of faults detected by instruction duplication. The third column indicates the percentage of simulations finished with a wrong output (with an undetected fault). The fourth column shows the percentage of escapes, that is, undetected faults that did not propagate to the output. The subsequent columns demonstrate the maximum, minimum, and average observed output corruption percentage due to undetected faults.

Table 4.2 demonstrates that no undetected faults damage more than 0.64% of the output. For most benchmarks, the maximum output damage is even smaller. The maximum output corruption of 0.06% in Image Addition means that in the output image, 1152 pixels out of almost 2 million are wrong. A visual inspection of the output image did not reveal any difference with the correct image. We consider the observed output damage in other benchmarks also tolerable.

SAD, Fibonacci, the JPEG encoder and the ADPCM decoder do not have any output damage due to undetected faults. All faults in these benchmarks are either detected or do not propagate to the output (escapes). These benchmarks have only a few instructions with low IVF values (see Section 4.3.3 and Figure 4.2). Most of their instructions have high IVF values, thus they are duplicated in IVF-SID, and faults are detected. This high instruction coverage results, however, in a smaller performance improvement of IVF-SID compared

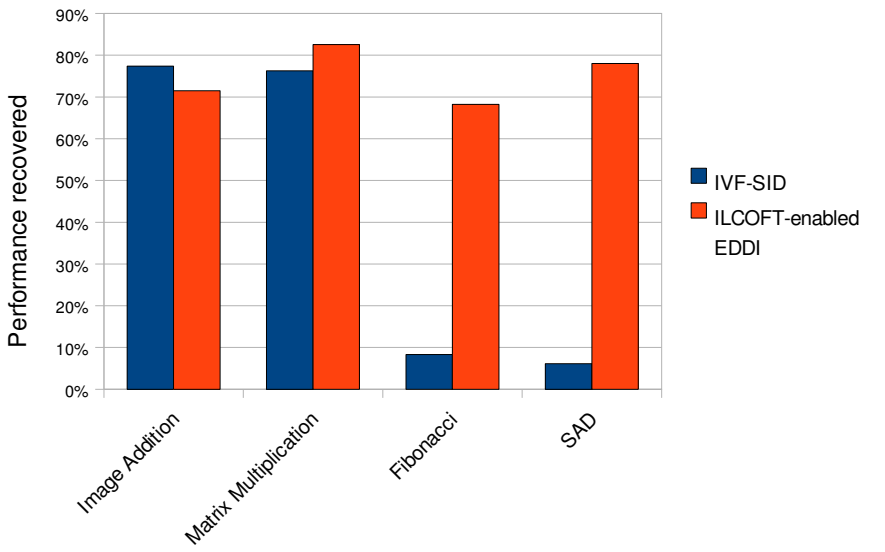


Figure 4.6: Comparison of performance penalty reduction achieved by IVF-SID over instruction duplication and by ILCOFT-enabled EDDI over EDDI.

to full instruction duplication.

4.3.5 IVF-SID Compared to ILCOFT-Enabled EDDI

To compare the manual instruction protection level assignment for ILCOFT with the automatic assignment based on IVF, this section compares ILCOFT-enabled EDDI described in Chapter 3 to IVF-SID. Figure 4.6 compares the amount of performance recovered by IVF-SID and by ILCOFT-enabled EDDI. Table 4.3 compares the maximum and average output damage observed when these techniques are used. ILCOFT-enabled EDDI data is taken from Chapter 3. ILCOFT-enabled EDDI duplicates only the critical instructions. Instructions are categorized manually by a programmer, and are considered critical if they affect the control flow.

Note that this is not a straightforward comparison. First, these techniques are very different, one works in hardware, while the other is applied in software. Second, the processor organizations used in this chapter and in Chapter 3 differ in some parameters such as memory access latency, cache sizes etc. Both organizations, however, have an issue width of four and four integer ALUs.

Table 4.3: Output damage of IVF-SID and ILCOFT-enabled EDDI.

	IVF-SID		ILCOFT-enabled EDDI	
	Maximum	Average	Maximum	Average
Image Addition	0.06 %	0.05 %	0.13 %	0.01 %
Matrix Multiplication	0.25 %	0.25 %	99 %	3.1 %
Fibonacci	0	0	96.67 %	38.23 %
SAD	0	0	100 %	100 %

The main message Figure 4.6 and Table 4.3 illustrate is that IVF-based ILCOFT is able to compete with the manual ILCOFT method. This is clear from the Image Addition and Matrix Multiplication performance and fault coverage: IVF-SID slightly improves the performance of Image Addition and reduces its average output damage, while it reduces the performance of Matrix Multiplication and improves its fault coverage. For Fibonacci and SAD, however, IVF-SID has a serious performance disadvantage. This is due to the fact that ILCOFT-enabled EDDI considers only instructions affecting the control flow to be critical. This leads to many undetected faults causing an average output corruption of 38.2% for Fibonacci and 100% for SAD. IVF-SID does not allow any undetected faults in these two kernels to reach the output, because it protects all the vulnerable instructions. Thus, IVF-SID is more suitable if reliability is the primary concern, and ILCOFT-enabled EDDI is preferable if performance is more important. Furthermore, the most important advantage of IVF-SID is that it does not require the programmer to assign the necessary protection level to every assembly instruction manually, which demands a serious effort and is very error-prone.

4.4 Conclusions

This chapter introduced the concept of Instruction Vulnerability Factor (IVF). IVF determines how much of the final application output is corrupted due to faults in particular instructions, and can be estimated using fault injection experiments. It is shown that in most applications, instructions have different IVF values. Depending on the nature of the application, it may have a large number of instructions with low IVF values, which means that faults in these instructions are tolerable. This work proposes to use the ILCOFT principle based on the application profiling data containing every instruction's IVF value. Instructions with higher IVF values are protected better than instructions with lower

IVF values.

A selective hardware instruction duplication scheme controlled by the instruction's IVF values (IVF-SID) was evaluated from the performance and fault coverage points of view. It is shown that, like in other ILCOFT schemes, both the performance and fault coverage of IVF-SID depend on the nature of the application. Like other ILCOFT schemes, IVF-SID recovers maximum performance (and also saves energy) in applications that produce many independent output elements. For these applications, faults in many instructions are likely to affect only a small part of the final output, and thus many instructions have small IVF, and are not protected. The minimum performance is recovered in applications producing a single output value, or a set of dependent values. Any fault propagating to the final output corrupts it completely in these applications, thus most of the instructions need to be protected. On the other hand, these applications have a better fault coverage, because most instructions are protected.

The experimental results demonstrate that for both these types of applications, IVF is a useful metric which allows to balance the performance and fault coverage. Moreover, IVF can be estimated automatically. It releases application developers from the need to attribute code for ILCOFT manually. This saves time, effort, and improves reliability, because manual instruction FT level assignment is very error-prone. IVF-based ILCOFT achieves performance improvements comparable to those of a manual ILCOFT method. In addition, as the fault coverage comparison in Section 4.3.5 shows, IVF-based ILCOFT provides much more accurate results than approaches based on the assumption that only instructions affecting the control flow are critical. Thus, IVF-based ILCOFT is much more accurate than other automatic (compiler-based) instruction attribution methods discussed in Chapter 3.

For future work, we plan to enhance our experimental setup. Currently there is the following limitation. Both in IVF estimation and fault injection, only the first dynamic occurrence of every static instruction is processed, because it is unknown if this instruction will ever be executed later. In most cases this is not problematic. In rare cases, however, this can lead to inaccurate IVF estimation. Consider, for example, an instruction which calculates a condition for a branch controlling a loop. If it produces any non-zero value, the subsequent branch is taken (or not taken). If at one loop iteration this instruction should produce a `1`, which becomes any other non-zero number due to a fault, the branch still performs correctly. However, at another loop iteration, this instruction should

produce a `0`, which becomes a non-zero value due to a fault. In the latter case the branch behaves in the wrong way. Taking into account the dynamic application behavior would solve this problem when estimating IVF.

A possible future profiling enhancement is to use a timer to detect cases when faults lead to a significant performance degradation. During IVF profiling, we have observed several cases when due to a fault, a critical loop iterates many orders of magnitude more times than it is supposed to. This leads to a corresponding performance degradation. If such performance degradation is unacceptable or if a significant profiling duration increase is unacceptable, a timer can be used to detect these cases. If the application execution takes more time than permitted, it can be terminated, assigning the maximum IVF value to the instruction where the fault was injected.

Both problems mentioned above can be solved by the following approach. All the instructions affecting the control flow can be assigned the maximum IVF, because they can always lead to application crashes. Then, the IVF is estimated using profiling only for the remaining instructions. The instructions affecting the control flow can be identified by the compiler.

IVF estimation as presented in this chapter is suitable for applications with equally important output values. However, there exist applications whose output consists of different parts of varying importance. For example, in a video sequence, bytes defining individual pixel values are less important than bytes defining frame attributes. For such applications the IVF estimation procedure has to be adapted to take into account the importance of individual output elements. This can be achieved by assigning different weight to the corruption of more and less important output elements.

Note. This chapter is based on the following paper:

Demid Borodin and B.H.H. (Ben) Juurlink, **Protective Redundancy Overhead Reduction Using Instruction Vulnerability Factor**, *Proceedings of the ACM International Conference on Computing Frontiers*, May 2010.

5

Instruction Precomputation and Memoization for Fault Detection

Instruction precomputation [19] and memoization [21, 22, 92] are performance improvement techniques based on result reuse. When possible, they avoid instruction execution by using already available results. These results are obtained during off-line profiling in the case of precomputation, and by previous instruction executions with memoization.

This chapter focuses on improving the performance and fault coverage of the instruction duplication error detection method using instruction precomputation. Moreover, it shows that precomputation combined with memoization is much more powerful than either technique in isolation. This is true in the non-redundant case (without error detection), and is especially evident when used with instruction duplication.

This chapter is structured as follows. Section 5.1 gives an introduction and describes related work. Section 5.2 presents the organization details of the discussed schemes. Section 5.3 presents and discusses the experimental results. Finally, conclusions are drawn in Section 5.4.

5.1 Introduction

As described in Chapter 2, Franklin [69] proposed to duplicate instructions in the pipeline to detect errors in FUs, internal buses between the dynamic scheduler and the FUs, and some errors in the dynamic scheduler. Every decoded instruction is duplicated in the dynamic scheduler of a superscalar processor. The instruction and its duplicate are then scheduled and executed in the regular way, and their results are compared. Although in general the slowdown is less than the intuitively anticipated 100% on a superscalar processor, this

method still incurs a significant performance penalty (on average 30% to 40% in our experiments) and covers mostly short transient faults. If redundant instructions execute on the same FU, this scheme in its pure form does not cover long-lasting transient, permanent, nor common faults in time. Furthermore, if the copies are executed on different FUs, common faults can still affect both of them in the same way. By long-lasting transient faults we mean faults that are present longer than one clock cycle. Common faults are faults affecting two different FUs in the same way, or (in time) the same FU at different times.

Parashar et al. [76] used the instruction memoization technique to improve the performance of the instruction duplication scheme based on [74]. Memoization (or memoing), also called instruction reuse, avoids redundant computations by reusing the result(s) of previous executions. The concept was introduced by Michie [20]. Memoization is traditionally used in software, manually or automatically. For example, a programmer can manually reuse the results of a previous function invocation. A look-up table with the function return value(s) corresponding to specific sets of input arguments is created at runtime, and is consulted on future invocations. Figure 5.1 illustrates a possible structure of such a function. Automatic memoization is used, for example, for

```
function ( arguments )
{
  if the result for the given arguments is available
    reuse the result and quit;
  perform computations;
  save the result and arguments in the look-up table;
}
```

Figure 5.1: Manual memoization example.

parsing in compiler technologies [146]. Memoization can be applied at different levels. For example, at the FU level, the FU reuses its previous results when the inputs match. At higher levels, results of an instruction, a block of instructions, or a high-level programming language function can be reused. This work focuses on instruction-level memoization, which reuses the results of instructions with matching operand values.

Several works proposed hardware schemes utilizing memoization. Richardson [21] proposed to save the results of non-trivial floating-point operations (with operands other than `0.0` and `1.0`, for example) in a *result cache*. The cache is direct-mapped and indexed by hashed operand values. The result cache is accessed in parallel with an executing floating-point operation. If a hit occurs, the result is reused and the full operation is canceled to save time.

Oberman and Flynn [147] addressed reducing the latency of the floating-point division operation. They proposed *division caches*, which are similar to result caches except that only the results of divisions are stored, and *reciprocal caches*. By saving the reciprocals of the divisors, the reciprocal caches convert the high-latency division operations to lower-latency multiplication operations. Sodany and Sohi [22] used memoization for all instructions rather than only for long-latency operations. To increase the benefit for single-cycle operations, they indexed the *reuse buffer* using the program counter (PC) instead of the instruction operand values. This way the reuse buffer can be accessed earlier in the pipeline, even when the operand values are not yet available. Citron et al. [92] used memoization for multimedia applications and focused only on long-latency instructions.

Parashar et al. [76] avoid the execution of duplicate instructions by reusing previously computed (memoized) results. Every executed instruction stores its input operands and output result in a special hardware buffer (which we call the *memo-table*). Subsequent original instructions are always executed normally, while duplicates perform a memo-table lookup and reuse the result, if available, instead of re-executing. The result of the original instruction is compared to the re-executed or reused result of the duplicate instruction. Later Gomaa and Vijaykumar [77] used memoization to verify only the instructions that hit the memo-table, leaving other instructions unprotected or (when it does not degrade performance) protected by another fault detection technique.

In this chapter we propose to use the *instruction precomputation* technique [19] (further referred to as *precomputation*), and a combination of instruction precomputation with memoization, to improve performance and fault coverage of instruction duplication. Precomputation is a work reuse technique involving off-line application profiling. The profiling data (instruction opcodes with operands and corresponding results) is stored together with the application binary code. Prior to execution, the profiling data is loaded into the *precomputation table* (*P-table*). When an instruction is about to be executed, a P-table lookup is performed. In the original (performance-oriented) precomputation scheme [19], if the instruction with the same input operands is found, the result is reused and the instruction is not executed. In the proposed (FT-oriented) scheme, the instruction is still executed, and the computed result is compared to the result from the P-table (similar to how memoization is used in [76]). Alternatively, if the profiling data and the P-table are sufficiently reliable, the precomputed result does not need to be re-computed.

Memoization as in [76] improves the fault coverage of instruction duplication

by covering more long-lasting transient and common faults. This is due to the time interval which memoization inserts between the first instruction execution (when the result is memoized) and the subsequent execution (when the result is compared to the memoized one). Long-lasting faults that disappeared or appeared during this interval, and thus did not affect both executions, are covered. Precomputation further improves the fault coverage of memoization by addressing all these faults, because the profiling is done at a different time and most likely even on a different system. In addition, for several benchmarks the precomputation-based scheme outperforms the memoization-based scheme.

Memoization exploits local instruction redundancy, while precomputation addresses globally dominant instructions. Memoization performs best when many instruction instances with the same inputs are executed closely in time. However, if redundant instruction executions are far away from each other, memoization might evict the previous result from the memo-table before the next instruction appears. Precomputation serves instructions that are globally dominant, but spread across the whole application execution, better, because the P-table holds these instructions and does not evict them.

To exploit the individual advantages of both precomputation and memoization, an additional contribution of this chapter combines them, achieving higher overall performance. Precomputation serves the globally dominant instructions, leaving more space in the memo-table for the locally dominant instructions. This eliminates conflicts between the global and local redundancy, exploiting both of them as much as possible. Experimental results demonstrate that without instruction duplication, precomputation combined with memoization most of the times outperforms both precomputation and memoization used alone. With instruction duplication, the advantage is even larger, because instruction duplication increases the pressure on the FUs. Instruction duplication enhanced with both precomputation and memoization reduces the performance degradation of duplication with either precomputation or memoization by on average 27.3% and 22.2%, respectively. The total hardware overhead is similar, because we use half-sized memo- and P-tables for the combination of memoization and precomputation. The performance advantage of the combined scheme over the precomputation scheme comes at the price of a reduced long-lasting fault coverage, because the P-table size halves, and thus fewer instructions hit it. In total, however, the combination protects more instructions by either precomputation or memoization, improving the coverage of shorter transient faults. Compared to the memoization-based scheme, the combination improves the long-lasting and permanent fault coverage, because some instructions are covered by precomputation, and the total instruction coverage

is similar.

To summarize, the main contributions of this chapter are:

- Instruction precomputation is used to improve the performance and fault coverage of instruction duplication. Every instruction executes at least once, and is either executed again or reused from the P-table. The output results are compared.
- An additional scheme is proposed which does not execute instructions that hit the P-table at all. The off-line profiling is assumed to be performed on a highly reliable system, and the P-table is assumed to be protected with ECC.
- Instruction precomputation is combined with memoization to improve the performance and fault coverage of both techniques used alone.

5.2 System Organization

The instruction duplication scheme used in this chapter has already been presented in Chapter 4 (Section 4.2.2). This section presents the implementation details of the other discussed techniques: precomputation (Section 5.2.1), memoization (Section 5.2.2), and instruction duplication extended with these techniques (Section 5.2.4). Section 5.2.3 discusses the table structure both for precomputation and memoization. The proposed combination of precomputation and memoization is discussed in Section 5.2.5.

5.2.1 Instruction Precomputation

Instruction precomputation avoids the re-execution of the most frequent instructions by using profiling information collected off-line. It was proposed by Yi et al. [19] to increase ILP. At the profiling stage, the executed instructions with their unique combination of input operands and output results (further called *instruction instances*) are sorted based on their frequency and included in the application binary code. Prior to the application execution, the profiling data is loaded into the P-table. Figure 5.2 visualizes the precomputation scheme at run time. An instruction proceeds regularly until the Execute stage. Then a P-table lookup is performed. If the necessary result is found, it is reused, and the instruction writes back on the next clock cycle. Otherwise, the instruction is issued to a corresponding FU, when it is available.

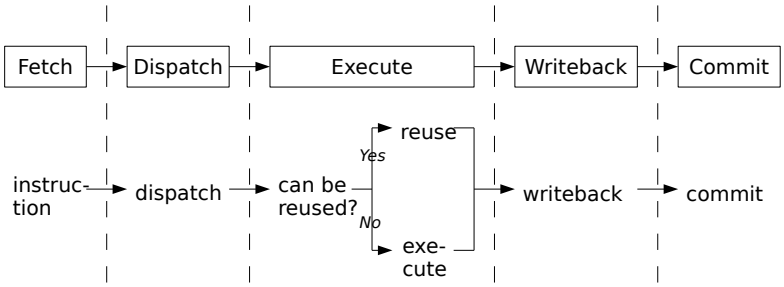


Figure 5.2: Instruction precomputation.

Note that unfortunately the P-table lookup cannot be performed before the Execute stage. This is because the instruction operand values are needed, and they are not guaranteed to be available at the earlier stages. This means that every instruction that hits the P-table has a single-cycle execution latency. For this reason, multi-cycle operations benefit more from precomputation (and also memoization) than single-cycle operations [92]. But in some cases even single-cycle operations benefit from precomputation and memoization. This happens when resources become a constraint and the issue of a single-cycle operation is stalled because no appropriate FU is available. Result reuse eliminates the need of FUs for instructions that hit the table, reducing the pressure on the resources.

The P-table is a hardware buffer of a limited size. It most probably cannot store all instruction instances seen in a relatively large application. A very large P-table would consume too much of the chip area, but would not be very efficient, because the advantage of storing infrequent instruction instances can be expected to be negligible. The goal is therefore to fill a relatively small P-table with the most useful instruction instances. Thus, to reduce the application storage requirements, only a part of the collected profiling information can be actually stored with the program text. It is only needed to guarantee that there is a sufficient number of instruction instances to fill the P-table as much as possible. This can be achieved by sorting the instructions in the profiling data using a certain criterion which determines how beneficial they are, and filling the P-table in that order.

The straightforward way is to sort the instruction instances by their frequency of occurrence. Then, only the most frequent instructions occupy the space in the P-table. However, a single-cycle instruction with a slightly

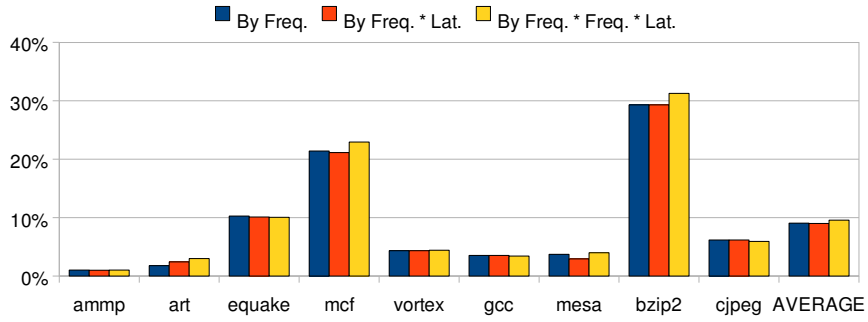


Figure 5.3: Performance comparison of three sorting methods of instructions in the precomputation profiling data. Average IPC increase of different precomputation configurations over the application execution without precomputation.

higher frequency than a multi-cycle instruction can be less beneficial in the P-table, because every hit will save fewer clock cycles. Thus, it is desirable to take into account not only the execution frequency, but also the latency. Figure 5.3 compares three different sorting methods: frequency-only (by $\mathit{frequency}$), equal priority (by $\mathit{frequency} \times \mathit{latency}$), and frequency-priority (by $\mathit{frequency} \times \mathit{frequency} \times \mathit{latency}$). The bars depict the average Instructions Per Cycle (IPC) increase of different precomputation configurations (described in Section 5.3) over the original application execution (without precomputation). The experimental setup is described in detail in Section 5.3. Figure 5.3 shows that the frequency-only and equal priority sorting methods never improve the IPC over 4.2% (cjpeg) more than the frequency-priority sorting does. The IPC improvement with the frequency-priority sorting is by up to 67.3% (art) larger than with frequency-only sorting, and by up to 35.5% (mesa) larger than with equal priority sorting. For these reasons, the frequency-priority sorting is used in this work.

5.2.2 Instruction Memoization

In our implementation, the memoization-based scheme is similar to the precomputation-based scheme discussed in Section 5.2.1, with a memo-table in place of the P-table. The main difference is that instead of performing off-line application profiling, memoization populates the memo-table dynamically while executing. Precomputation never updates the P-table after it is filled by the program loader. Memoization updates the memo-table at the Writeback

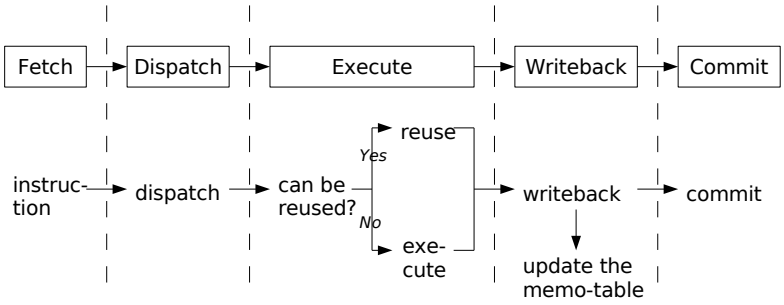


Figure 5.4: Instruction memoization.

stage for every executed instruction, evicting old data if necessary. In our implementation, the memo-table uses the Least Recently Used (LRU) replacement policy when evictions are required. Figure 5.4 visualizes the instruction memoization scheme.

Only computational instructions (integer and floating-point) are reused by both precomputation and memoization. Other instructions such as memory accesses are not. Memory stores cannot be verified by the reuse methods, because they do not produce any output values. Loads cannot be reused, because the loaded value may differ at different times. Note, however, that a memory access instruction is split in two parts: address calculation and the memory access itself. The address calculation part, which is an integer operation, can be reused.

5.2.3 Table Structure

One of the important precomputation and memoization design decisions is the structure of the P- and memo-table, since it needs a fast access time (one clock cycle in our organization). Structuring the table as a single large array with instruction instances (as they appear in the sorted precomputation profiling list) is not feasible. It would require a sequential access to every instruction instance in the table and comparison against the requested instruction, which is very time-consuming for relatively large tables.

In [78] we used a rather complex table structure, which indexed the P-table by instruction opcodes. We preferred this to indexing by a subset of input value bits (as some previously proposed memoization schemes do), because the opcode is often available earlier than the input values. The access to a table indexed by the opcodes can be started at the instruction decode stage,

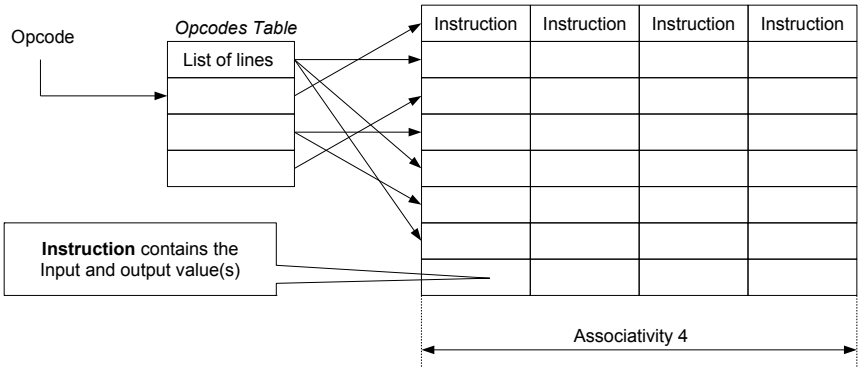


Figure 5.5: Precomputation table structure used in [78].

and finished when the input values are known. Indexing by the opcode rather than the PC as in [22] also allows different static instructions to reuse each other's results [92].

The P-table used in [78] contains cache-like *sets* holding a number of *entries* with instruction input and output values. The number of entries per set is determined by the associativity. All the entries in a set correspond to a single opcode. The experimental results show that among the most frequent instructions in the profiling data, some opcodes significantly dominate others. Thus, the table structure should allow an uneven instruction distribution. This can be achieved by assigning multiple P-table sets to a single opcode. This P-table structure is shown in Figure 5.5. The *opcodes table* holds a list of table sets assigned to the corresponding instruction opcode. When accessing the P-table, the opcodes table is consulted for the list of sets holding instruction instances with the required opcode. Then these sets are searched (preferably in parallel to reduce the access time) for the required input values combination. Note that for commutative arithmetic operations different combinations of the input operands can be checked to improve the hit rate and/or reduce the table size. Separate P-tables (or sets) might be used for instructions with operands of different size to optimize the resource usage.

The possible number of P-table sets assigned to a single opcode is limited. This is because the space in the opcodes table allocated for the list of used sets is limited. In addition, the requirement to provide a fast (fixed-latency) P-table access, which searches all the sets (preferably in parallel), limits this number. On the other hand, the more sets per opcode are allowed, the more instruction

instances with the most frequent opcodes are likely to be stored. Reducing this limit severely increases the risk to fill the P-table with less useful instructions, and thus to decrease the overall benefit in performance and fault coverage.

Note that the P-table lookup performed before issuing an instruction to a FU could appear on the critical path and affect the cycle time. If this happens, certain techniques can diminish or solve the problem. For example, the P-table access can be started at earlier stages, because the instruction opcode is already available after decoding. Then, the lookup is finished when the operand values are available. Moreover, the instructions producing the input operands could initiate the P-table lookup as soon as the values are available. In [22], the authors also argue that memo-table accesses (which are similar to P-table accesses) are unlikely to be a serious problem.

The memo-table we used in [78] is similar to the P-table described above. However, the memo-table differs in that it does not have the opcodes table, and thus, only one set per opcode is available. This is because unlike the P-table, the memo-table needs to be updated very often (about every cycle, possibly for multiple instructions). This would require complex logic, increasing the access time and cost. We do not consider it feasible to update the memo-table with a complex structure including the opcodes table so frequently. The memo-table uses the LRU replacement policy for the entries and the sets (to choose which set to assign to another opcode when needed).

Subsequent research revealed that the difference between the described P- and memo-table structure noticeably influences the performance achieved by memoization. The absence of the opcodes table in the memo-table leads to an even opcodes distribution. However, as mentioned above, some opcodes significantly dominate others and need more space in the table. Moreover, due to some rare opcodes, not all the table space is used in the opcode-based P- and memo-table configuration. This is because every table set can only hold instructions with the same opcode. Figure 5.6 shows the P-table occupation for different configurations (for every benchmark, the table size increases from left to right). For most applications larger tables provide a better utilization percentage, because the same number of empty table slots represent a smaller part of the whole table. For all applications smaller associativity leads to a better table occupation, because fewer instruction instances with rare opcodes are needed to fill all the entries in the set.

To reduce the table structure complexity and solve the problems discussed above, we reverted to the table indexing based on instruction input values and

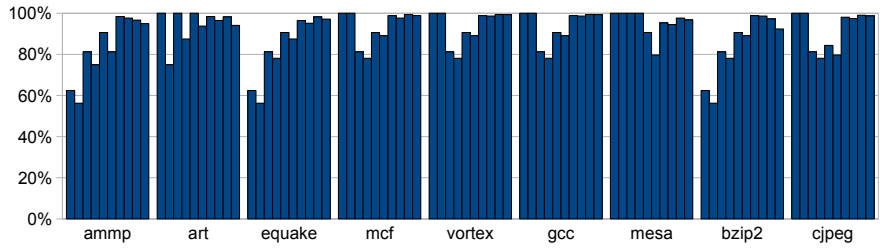


Figure 5.6: Precomputation table utilization (% of the whole table). P-table configurations (# sets, associativity) for every benchmark, from left to right: (2,4), (2,8), (4,4), (4,8), (8,4), (8,8), (64,4), (64,8), (128,4), and (128,8).

tried to improve it in [93]. A possible solution is to perform the exclusive OR (XOR) operation on the input operand values, and use (a subset of) the result bits to index the table. The problem of the XORed operand values is that the table cannot hold different instructions with the same popular input values. This can be solved by subsequent XORing the result with the instruction opcode, and thus assigning different P-table sets for these conflicting instructions. Figure 5.7 compares the performance of three P-table indexing strategies: by opcode (the table structure used in [78] and discussed above), by XORed input operand values, and by XORed input operand values and the instruction opcode. Figure 5.7 shows that the opcode-based indexing used in [78] outperforms the XORed operands-based indexing, achieving on average 8.2% higher IPC increase. However, XORing the operands with each other and with the instruction opcode achieves better performance results than opcode-based indexing, improving the IPC increase more, by on average 15.4%.

Thus, further in this chapter, both the memo- and P-tables are indexed by the XORed instruction operands and opcode (the result is taken modulo the number of sets in the table). The cache-like tables consist of sets, each holding a number of instructions with their input operands and output values. The number of instructions per set is defined by the table associativity. A P-table with N sets and associativity equal A will be further referred to as *P(N,A)*. The memoization scheme with a similar memo-table organization will be referred to as *M(N,A)*.

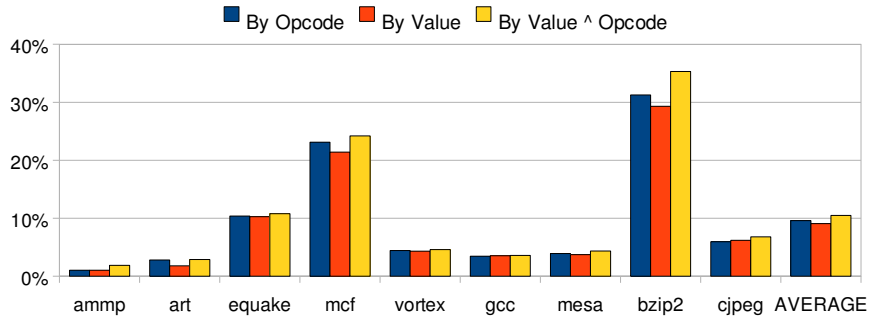


Figure 5.7: Performance comparison of three P-table indexing strategies: by opcode, by XORed instruction operand values, and by XORed instruction operands and opcode. Average IPC increase (for the table configurations as in Figure 5.6) over execution without precomputation.

5.2.4 Duplication with Precomputation or Memoization

In this work instruction precomputation and memoization are used in combination with instruction duplication. Figure 5.8 depicts the scheme employing both instruction duplication and memoization (*D+M*). An instruction is always executed at least once, and executed once more if the result is not found in the memo-table. The computed result is then compared to the reused or recomputed result at the Writeback stage. Successful instructions update the memo-table and commit.

The duplication with precomputation scheme (*D+P*) could be similar to D+M (Figure 5.8). However, precomputation differs from memoization in that the instruction results are computed off-line, only once per application. This suggests that if the profiling data is gathered on a highly-reliable host system and is protected well, it can provide the reliability of the profiling system on the target system. Protecting the P-table with ECC and some control logic redundancy would be sufficient for that. The error correcting capability of the P-table would even bring a partial recovery capability into otherwise fail safe-only duplication-based system. Given a sufficiently reliable profiling system and P-table, the execution of the instructions that hit the P-table can be skipped altogether without any reliability loss. This scheme is referred to as *duplication with precomputation only* (*D+PO*). In [78] we show that D+PO significantly outperforms D+P with original instruction re-execution (as in D+M). Thus, in this chapter, only D+PO is used, and D+P further refers to D+PO. D+M always executes the instructions at least once (otherwise the reliability would

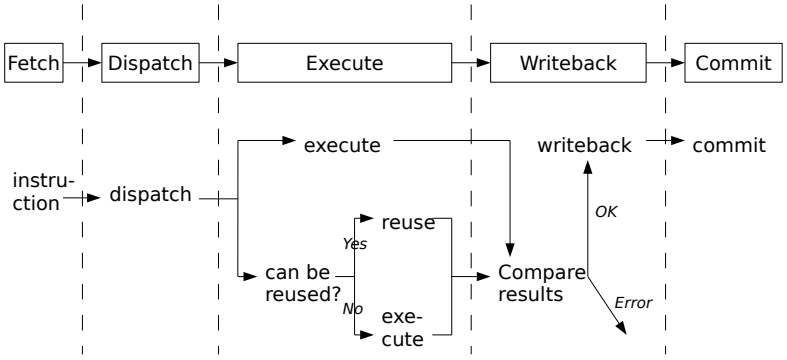


Figure 5.8: Instruction duplication with memoization (D+M).

have suffered), while D+P (D+PO) does not execute instructions that hit the P-table at all.

5.2.5 Precomputation Combined with Memoization

In addition to using precomputation for fault detection (D+P), this chapter focuses on the combination of precomputation and memoization (*P+M*). P+M applied to the duplication-based scheme is further referred to as *D+P+M*.

Figure 5.9 depicts the P+M scheme. P+M assumes that the application has been profiled off-line for precomputation. At run time, the profiling data is loaded to the P-table, and precomputation works in the regular way. Precomputation has priority over memoization. Instructions hitting the P-table are reused and are not written to the memo-table. Writing instructions hitting the P-table to the memo-table is likely to evict useful data, but has no benefit since these instructions remain in the static P-table for the whole program execution. The memo-table is reserved only for instructions missing the P-table. These instructions perform a memo-table lookup. Instructions that hit it are reused, and the others are executed. This organization allows precomputation to serve the globally dominant instructions, and leaves as much space as possible in the memo-table for the locally redundant instructions. For performance reasons, the P- and memo-table lookups are performed in parallel. If neither one hits, the instruction is executed normally.

Figure 5.10 presents the D+P+M scheme. The memo-table update step at the Writeback stage, which is similar to that in Figure 5.9, is omitted for readabil-

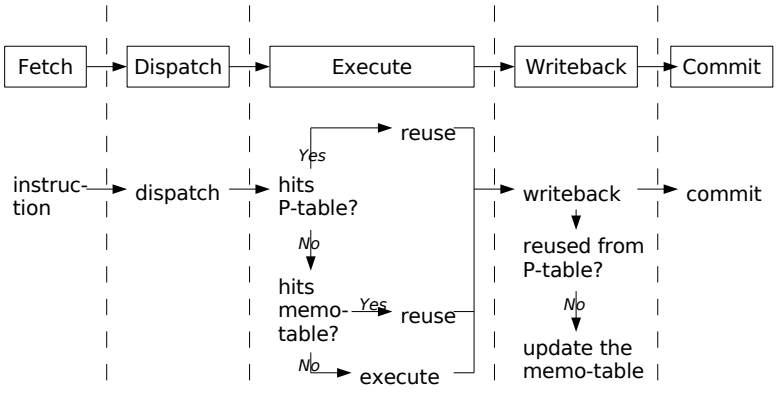


Figure 5.9: Instruction precomputation with memoization (P+M).

ity. Instructions hitting the P-table are considered sufficiently reliable. They progress to Writeback and Commit without verification. Other instructions are always executed once and are either reused (from the memo-table) or executed another time. The results are compared at the Writeback stage, and an error is signaled on mismatch.

In this work, when comparing P+M-based schemes with the precomputation and memoization-based schemes, the size of both the memo- and P-table is always halved in P+M. In other words, the *P+M(N,A)* scheme has a memo- and a P-table each with $\frac{N}{2}$ sets, of the associativity A. Thus, while *P(N,A)* has a single table with $N \times A$ entries, *P+M(N,A)* has two tables each of the size $\frac{N}{2} \times A$. This ensures that the hardware overhead is similar in the compared schemes.

5.3 Experimental Results

This section evaluates the performance and fault coverage of the duplication with precomputation (D+P) scheme, as well as the combination of precomputation and memoization (P+M and D+P+M). P+M is compared to precomputation and memoization, and D+P+M is compared to D+P and D+M.

Several integer and floating-point benchmarks from the SPEC CPU2000 suite [148, 149] are used, as well as the JPEG encoder multimedia application. Since with precomputation applications are unlikely to have the same

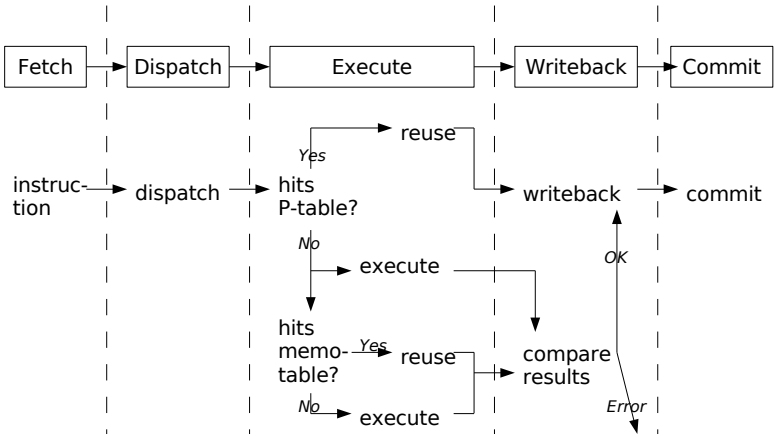


Figure 5.10: Instruction duplication with precomputation and memoization (D+P+M). Memo-table update at Writeback (as in Figure 5.9) is skipped.

input at the profiling and deployment stages, different inputs are used at these stages.

Section 5.3.1 describes the used simulation platform. Section 5.3.2 discusses the fault coverage of the considered duplication-based schemes. Section 5.3.3 evaluates the performance of these techniques.

5.3.1 Simulation Platform

The experiments are performed using the SimpleScalar tool set [140]. We modified the *sim-outorder* simulator to model instruction precomputation, memoization, P+M, duplication, D+P, D+M, and D+P+M. The processor configuration used is presented in Table 5.1. In Section 5.3.3, when evaluating performance, the number of integer ALUs is varied to demonstrate how instruction reuse affects the throughput. The configurations (number of sets and associativity) of the memo- and P-tables are also varied. The benchmark applications were run, skipping the first 100 million instructions to bypass the I/O overhead, either until completion or until the next 100 million instructions committed.

To minimize the resource overhead and keep the cost low, only small to modestly-sized memo- and P-tables are examined in our experiments. The number of sets in the tables varies from 8 to 1024, and associativity from 1 (direct mapped) to 4. This means that the total memo-/P-table size varies from

Table 5.1: Processor configuration.

Fetch/Decode/Issue/Commit Width	8
Branch Predictor	Combined, 8K meta-table
BTB Size	1 K, 2-way associative
RUU Size	128
LSQ Size	64
# of Int. ALUs	1 to 4
# of Int. Mult./Div.	1
# of FP ALUs	1
# of FP Mult./Div.	1
Memory Latency	112 cycles (first chunk), 2 cycles (subsequent chunks)
L1 Data Cache	32 KB, 2-way set associative
L1 Instruction Cache	32 KB, 2-way set associative
L2 Unified Cache	512 KB, 4-way set associative

approximately 104 B to 52 KB (assuming that for every instruction the tables hold one 8-bit opcode and two 32-bit input and one output values, without protective information).

5.3.2 Fault Coverage

It is difficult to compare (quantitatively) the fault coverage of the considered duplication-based schemes. Therefore, two indirect methods are used to evaluate the fault coverage: the average memo-table instruction lifetime and the hit rate.

D+M improves the fault coverage of pure duplication by targeting some long-lasting transient, permanent, and common faults in FUs, data buses etc. D+M achieves this due to the time gap which it inserts between the execution of the memoized instruction and its subsequent (being verified) execution. Assume that an instruction result is memoized at time T_1. N clock cycles later it is compared to the recomputed result at time T_2. D+M covers the long-lasting faults that appeared before T_1 and disappeared between T_1 and T_2, or appeared between T_1 and T_2. Other long-lasting and permanent faults cannot be detected by D+M, because they affect both the memoized and recomputed results in the same way (if executed on the same FU). Hence, the number N determines

Table 5.2: Average memo-table instruction lifetime for different D+M configurations (% of the total execution time).

D+M config.	ammp	art	equake	mcf	vortex	gcc	mesa	bzip2	cjpeg
(2,4) to (8,8)	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
(64,4)	6.0	0.9	8.3	4.6	2.3	11.0	8.6	28.2	17.0
(64,8)	4.7	0.8	6.2	3.2	1.4	8.1	8.6	24.1	12.8
(128,4)	6.0	0.9	8.3	4.6	2.3	11.0	8.6	28.2	17.0
(128,8)	4.7	0.8	6.2	3.2	1.4	8.1	8.6	24.1	12.8

how efficient D+M is in covering long-lasting faults. The greater N is, the more long-lasting faults are likely to appear or disappear during this time. We call N the *memo-table instruction lifetime*, because it determines how long an instruction resides in the memo-table before it is reused. Note that pure duplication is also likely to insert a certain time gap between the redundant instruction executions (a gap of a few clock cycles can be expected). The advantage of D+M over the pure duplication depends on how much greater the memo-table instruction lifetime is than the duplication gap.

Table 5.2 presents the average memo-table instruction lifetime for different D+M configurations. The instruction lifetime is measured in clock cycles, and presented as a percentage of the total execution time. For the smallest configuration D+M(2,4) the average instruction lifetime is 3.5 (ammp) to 8.2 (gcc) clock cycles. This means that the long-lasting fault coverage of D+M(2,4) is comparable to that of pure instruction duplication, because a gap of 3 to 8 clock cycles can be expected between the execution of two redundant instruction copies in the duplication scheme. For all configurations from D+M(2,4) to D+M(8,8) the average instruction lifetime is up to 0.02% of the total application execution time. With larger memo-table sizes, the average instruction lifetime grows significantly, improving the long-lasting fault coverage of D+M. However, as Table 5.2 shows, it is between 1.4% and 28.2% of the total execution time even for the largest memo-tables.

D+M also simplifies the fault location/recovery of the duplication scheme. When two results do not match in the duplication scheme, either FU involved in the computation could be faulty. With D+M, the result held in the memo-table has already been verified by redundant computation before it was saved.

Thus, if the newly computed and the memoized results do not match, the FU on which the last computation was performed is most likely to be faulty. The situation corresponds to TMR [3]. This is, however, only true if the storage elements in the memo-table provide sufficient reliability, guaranteeing that the saved result is not corrupted when being reused. This can be achieved, for example, by protecting the memo-table with ECC.

D+P has all the advantages of D+M, and further improves the fault coverage by protecting against all the long-lasting transient, permanent, and common faults. This is due to the fact that the application profiling is performed at a different time than the actual execution, and most likely even on different hardware (host machine). Thus, permanent, long-lasting transient, and common faults cannot affect both the profiling and the actual execution results.

D+P+M improves the long-lasting fault coverage of D+M, because some instructions are covered by precomputation instead of memoization. D+P+M reduces the long-lasting fault coverage of D+P, because with the reduced P-table size, the number of instructions covered by precomputation diminishes. The following hit rate evaluation measures the fault coverage gain and loss of D+P+M compared to D+P and D+M.

Another indirect fault coverage measure is the memo- and P-table hit rates. The hit rate is defined as the ratio of table hits to the total number of executed instructions. It determines how many instructions are protected by precomputation or memoization in the D+P, D+M, and D+P+M schemes, and how many are only duplicated. Unlike performance, the hit rate does not vary significantly when the number of FUs in the system changes. The factors on which the hit rate depends are the memo- and P-table size and configuration. Thus, only the results for systems with 1 integer ALU are presented.

Figure 5.11 shows the precomputation hit rate, and Figure 5.12 memoization hit rate of different table configurations. The upper parts of the bars represent the fraction of reused multi-cycle instructions, and the lower parts the fraction of reused single-cycle instructions. Figure 5.13 presents the memo-table (the upper part of every bar) and the P-table (the lower part of every bar) hit rates in the P+M scheme. Figure 5.14 compares the hit rate (average for different table configurations) of precomputation, memoization and P+M. The bars for every benchmark in Figure 5.11, Figure 5.12, and Figure 5.13 represent the following configurations from left to right: *(8,1)*, *(8,2)*, *(8,4)*, *(16,1)*, *(16,2)*, *(16,4)*, *(64,1)*, *(64,2)*, *(64,4)*, *(256,1)*, *(256,2)*, *(256,4)*, *(512,1)*, *(512,2)*, *(512,4)*, *(1024,1)*, *(1024,2)*, and *(1024,4)*.

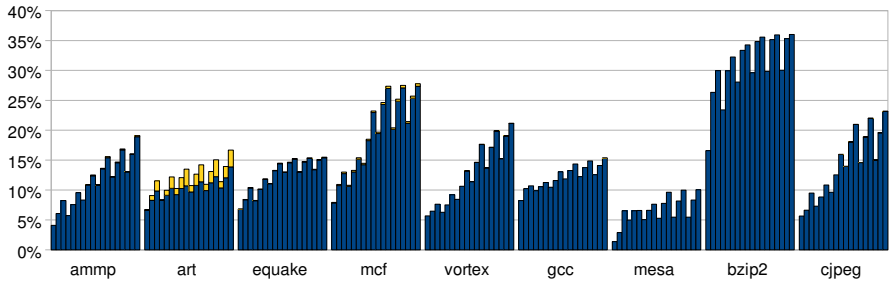


Figure 5.11: Precomputation hit rates for different table configurations. Upper parts of the bars represent the fraction of reused multi-cycle instructions, lower parts the fraction of reused single-cycle instructions.

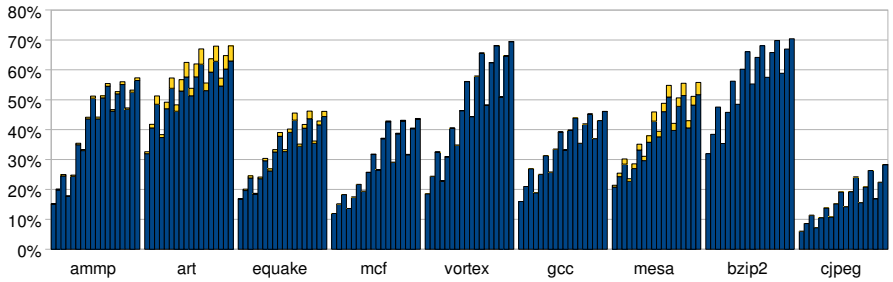


Figure 5.12: Memoization hit rates for different table configurations. Upper parts of the bars represent the fraction of reused multi-cycle instructions, lower parts the fraction of reused single-cycle instructions.

Figure 5.14 shows that for most benchmarks precomputation covers significantly fewer instructions than memoization. Thus, although D+P improves the long-lasting and permanent fault coverage of D+M, it reduces the total number of protected instructions. Figure 5.14 also shows that the average hit rate of the P+M scheme is only 0.9% less than the memoization hit rate. This means that approximately the same percent of executed instructions is protected by D+P+M and D+M. From Figure 5.13 it follows that on average 36.1% of the instructions in P+M (and thus also in D+P+M) hit the P-table, and the rest hit the memo-table. Thus, while about the same number of instructions is protected in the D+P+M and D+M schemes, the long-lasting fault coverage of 36.1% of these instructions is improved by precomputation in D+P+M. Figure 5.14 also shows that P+M hit rate is on average 2.6 times higher than that of precomputation. Thus, D+P+M protects significantly more executed instruc-

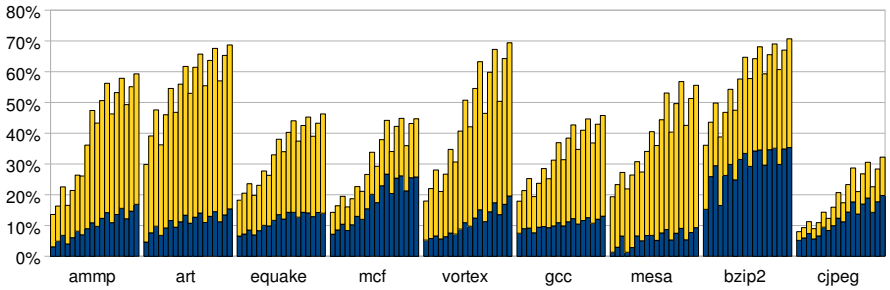


Figure 5.13: P+M hit rates for different table configurations. Lower parts of the bars represent the P-table, upper parts the memo-table hits.

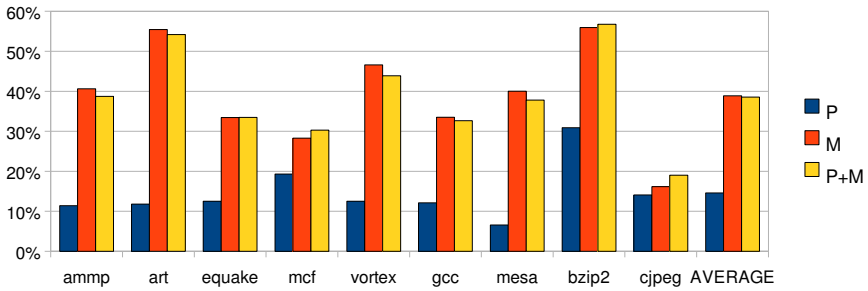


Figure 5.14: Comparison of the average hit rates for different table configurations in precomputation (P), memoization (M) and P+M.

tions than D+P, by either precomputation or memoization. However, D+P+M protects on average 10.6% less instructions by precomputation than D+P does (the others are covered by memoization). These instructions have a reduced long-lasting fault coverage.

5.3.3 Performance

Figure 5.15 compares the IPC increase of the different schemes over the original (non-redundant) execution on a system with 2 integer ALUs. It presents the average results over the different table configurations. Figure 5.16 compares the IPC increase for a single memo- and P-tables configuration with 512 sets and the associativity of 4 (256 sets in every table of P+M-based schemes). This table configuration will further be recommended as one of the most optimal alternatives. Duplication degrades performance, and thus has a negative

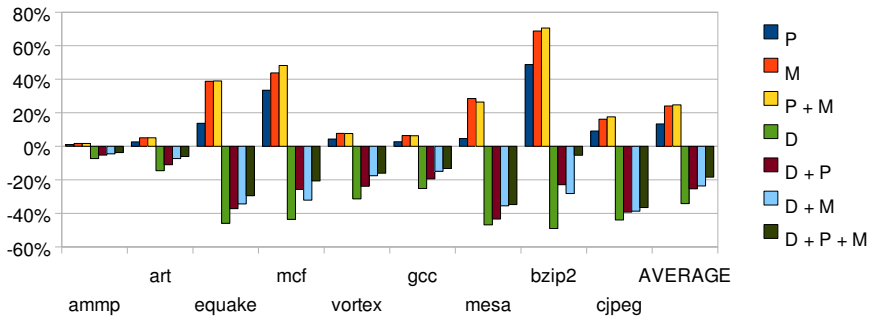


Figure 5.15: IPC increase of different precomputation (P), memoization (M), and duplication (D) schemes over the original execution on a system with 2 integer ALUs. Average for different table configurations.

IPC increase. Note that the shown IPC of the duplication-based schemes is the number of original instructions (without duplicates) divided by the execution time in clock cycles, thus it can be compared to the IPC of the non-redundant schemes.

The performance of P+M is for most benchmarks slightly higher than the performance of memoization (the IPC increase improves by on average 2.6%), and significantly higher than that of precomputation. D+P achieves better performance than D+M for some benchmarks, but performs slightly worse (has 6.5% less IPC increase) on average. D+P+M always outperforms D+M and D+P, reduces their performance degradation due to instruction duplication by on average 22.2% and 27.3%, respectively. We explain D+P+M advantage compared to P+M by the doubled FUs requirements of the duplication-based schemes. In addition, the fact that instructions hitting the P-table are not executed plays an important role: D+PO reduces the IPC by on average 9% less than D+P (which executes hitting instructions) does [78]. Moreover, in a few cases D+P+M even outperforms the original execution. For example, for mcf running on a system with 1 integer ALU, *D+P+M(512,4)* and *D+P+M(1024,4)* increase the IPC of the original (non-redundant) execution by 2.8% and 3.5%, respectively. On a system with 2 integer ALUs, bzip2 running on the configuration *D+P+M(512,4)* also outperforms the original (see Figure 5.16).

Both precomputation and memoization improve performance when the number of FUs is a system bottleneck. Thus, the performance improvement decreases when the number of integer ALUs increases. Figure 5.17 shows the

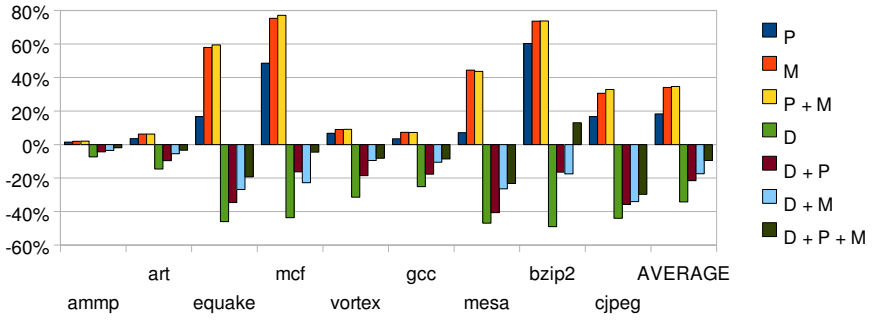


Figure 5.16: IPC increase of different precomputation (P), memoization (M), and duplication (D) schemes over the original execution on a system with 2 integer ALUs. P- and memo-tables have 512 sets and the associativity is 4.

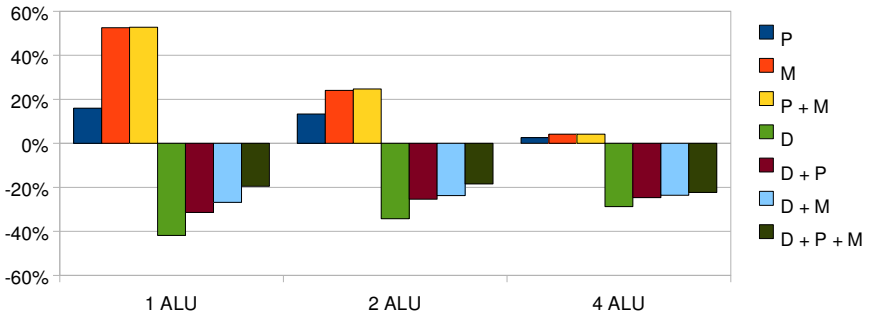


Figure 5.17: IPC increase of different schemes with different number of integer ALUs. Average for different table configurations and benchmarks.

average IPC increase (across all the benchmarks) of the considered schemes for varying number of integer ALUs in the system. With 1 ALU D+P+M reduces the IPC decrease, compared to the instruction duplication scheme, by 53.4%, with 2 ALUs by 46.1%, and with 4 ALUs by 22.2%. Compared to D+M, with 1 ALU D+P+M reduces the IPC decrease by 27.3%, with 2 ALUs by 22.2%, and with 4 ALUs by 5.2%.

Figure 5.18 shows how the performance of different schemes (across all the benchmarks) depends on the memo- and/or P-table configuration. To explore the limits, Figure 5.18 also includes a very large configuration with 64K sets and an associativity of 16 (corresponding to a table of approximately 13 MB). Figure 5.18 demonstrates the importance of the table associativity,

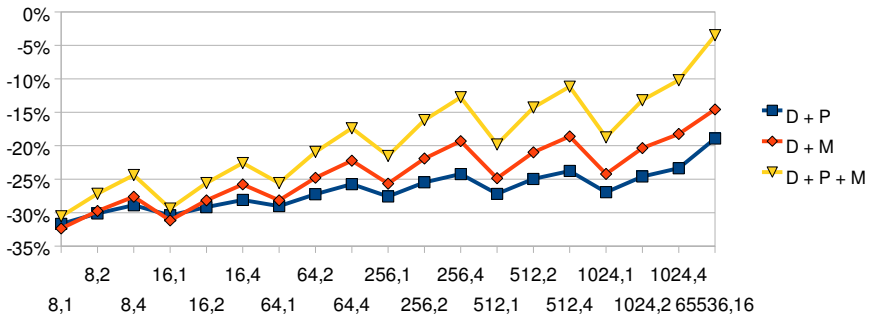


Figure 5.18: IPC increase for different table configurations. Average for systems with 1, 2 and 4 integer ALUS.

which significantly improves performance for all the configurations. For example, *D+P+M(16,4)* has significantly higher performance than *D+P+M(64,1)*, and *D+P+M(512,2)* outperforms *D+P+M(1024,1)*. Figure 5.18 also shows that the performance improvement diminishes when enlarging larger tables (with 256 and 512 sets) in comparison with the smaller ones. For example, *D+P+M(1024,4)* doubles the size of *D+P+M(512,4)*, while its IPC increase improves by less than 1%. Thus, *D+P+M(512,4)* might be a good candidate for the optimal D+P+M table configuration from the performance vs. hardware overhead point of view. However, the limit has not yet been reached, as *D+P+M(65536,16)* still significantly outperforms all the smaller configurations.

5.4 Conclusions

This chapter proposes and evaluates instruction precomputation for error detection purposes. Moreover, it binds the strengths of precomputation and memoization to achieve better performance than any of these techniques achieves alone. Precomputation focuses on the globally dominant instructions, while memoization exploits the local redundancy. Applied together, precomputation exempts memoization from the globally dominant instructions, allowing more space in the memo-table for the locally dominant ones. The advantage is especially prominent when this combination is used to improve the performance and fault coverage of the instruction duplication error detection technique.

D+P+M covers approximately the same number of instructions as D+M, but

introduces the full long-lasting and permanent fault coverage of on average 36.1% of them. At the same time, D+P+M reduces the IPC penalty of D+M by on average 22.2%. Compared to D+P, D+P+M covers on average 2.6 times as many instructions, but on average 10.6% less instructions are covered against long-lasting and permanent faults. Given the 2.6 times higher total instruction coverage and the average performance degradation reduction of 27.3% over D+P, the long-lasting fault coverage loss of 10.6% seems to be an acceptable price.

In this chapter, precomputation and memoization used equally sized tables to store instructions. In future we plan to investigate how memo- and P-tables of different sizes influence the performance and fault coverage of the combined scheme. We also plan to reduce the number of instructions appearing in both the P- and memo-tables by using trivial computation detection. In other words, instructions performing trivial arithmetic operations such as addition with `0` or multiplication by `1` or `0` will be detected and processed separately, they will not occupy valuable space in the memo- and P-tables.

Note. This chapter is based on the following papers:

Demid Borodin, B.H.H. (Ben) Juurlink, and Stefanos Kaxiras, **Instruction Precomputation for Fault Detection**, *DSD'2009: Proceedings of the 12th Euromicro Conference on Digital System Design*, pp. 91–99, August 2009.

Demid Borodin and B.H.H. (Ben) Juurlink, **Instruction Precomputation with Memoization for Fault Detection**, *DATE'2010: Proceedings of the Design, Automation and Test in Europe*, March 2010.

6

A Low-Cost Cache Coherence Verification Method for Snooping Systems

While the techniques presented in previous chapters have been evaluated on single processor systems, they can also be effectively applied to individual cores in multiprocessor systems. It is impossible, however, to completely protect multiprocessor systems with techniques from the unicore world, because multiprocessors introduce some unique reliability issues.

This chapter addresses one of the multicore-specific reliability issues, namely the cache coherence operation verification. Correctness of the cache coherence operation is crucial for multiprocessor systems supporting cache coherence. This chapter proposes a low-cost error detection technique for snooping-based cache coherence protocols. For the widely used MESI coherence protocol, the proposed method does not introduce any performance overhead. Only a limited amount of additional hardware is required. Existing systems can be easily extended to support the proposed technique. Almost all single faults that are able to affect data integrity in the system are covered, with the exception of a few very rare cases.

This chapter is organized as follows. Section 6.1 presents the motivation. Section 6.2 describes other existing cache coherence verification methods and compares them to the proposed one. Section 6.3 explains the proposed technique, discusses possible design options, and gives an example implementation for the MESI coherence protocol. Section 6.4 analyzes the fault coverage of the proposed method, and presents the experimental results. Finally, Section 6.5 summarizes this chapter.

6.1 Introduction

Current trends in computer architecture demonstrate an increasing interest in multiprocessor systems [128]. Multiprocessors feature a distributed and/or shared memory hierarchy. To enhance programmability, the details of the memory structure are often hidden from the application developer, for whom the memory structure appears as a single shared memory accessible from all the nodes. This is achieved by employing cache coherence protocols which guarantee a consistent memory view for different nodes [125].

Cache coherence plays a very important role in multiprocessors, since it provides data integrity. For example, it guarantees that when one processor has changed data at a certain memory address, other processors in the system get a fresh copy of these data when they read them. Thus, faulty cache coherence hardware can lead to data integrity violation. In the above example, if the cache coherence hardware fails to notify a reading processor about the changes another processor has made to the data, it will work with wrong (stale) data, produce other wrong data on this basis, etc.

This chapter proposes a concurrent error detection technique addressing cache coherence in multiprocessors. The concept of watchdog processors [126] is used. A watchdog processor is external logic that dynamically verifies the operation of controlled logic based on certain information received from it. The proposed technique addresses multiprocessors with snooping-based cache coherence. The operation of the coherence logic and storage elements holding coherence states in every cache is dynamically verified by external logic (*checker(s)*) located on other cache(s) or on special circuit(s) snooping the system network. Each checker has a local store with tag information for all the cache lines residing in the checked cache, but not the data. The checkers snoop all messages on the network and filter those related to the checked cache. Based on these messages and the tag information they contain, the checkers maintain the correct state of each cache line, and check the correctness of its state in the checked cache. Wrong states, which might result in violated data integrity in the system, are detected and signaled. In addition to this, the checkers partly verify the operation of the communication logic, making sure, for example, that every request is answered.

The main features that distinguish our proposal from the related approaches discussed in Section 6.2 are:

- Low or zero performance overhead. There is no performance overhead for the MESI protocol verification discussed in this chapter.

- Limited hardware overhead, comparable to the cheapest alternatives.
- The design is scalable in that the overhead is linear in the number of nodes in the system.
- It is relatively straightforward to extend an existing system with the coherence checking. The checkers have to be attached to the network, and a few bits have to be added to the messages. Cache controllers have to add necessary information to these bits.
- The benefits come at the price of a reduced fault coverage compared to some other approaches that introduce considerably more overhead.

6.2 Related Work

Several dynamic cache coherence verification techniques have been described in Section 2.6. Here we focus on the differences between these proposals and our proposal.

Cache coherence verification techniques differ in the achieved fault coverage and the introduced hardware overhead. All existing techniques introduce a certain hardware overhead needed for the additional logic. Furthermore, most techniques increase the communication network bandwidth requirements, which means that they introduce either a performance penalty or a hardware overhead (the capacity of the network needs to be extended). All techniques providing full fault coverage, which includes faults in the communication network such as the loss of coherence messages, introduce a certain bandwidth overhead. Other proposed methods, even those that do not provide full fault coverage, also introduce a certain bandwidth overhead.

Unlike previous proposals, our method introduces almost no network bandwidth overhead (only a few bits must be added to each message). This, however, is achieved at the expense of a limited coverage of network faults that can lead to global errors (conflicts in coherence states of different nodes). We assure that the communication network is reasonably robust. For example, if a coherence message is lost (neither reaches the addressed cache nor its checker), and the system does not detect this, our technique will only detect this fault if it triggers illegal coherence activities in future. However, some global conflicts can be detected from the messages received from other caches. For example, if a checker receives a flush request from a remote cache, it makes sure that the

corresponding line in the checked cache is in the shared state, otherwise it signals an error. In addition, in comparison with existing methods, ours requires minimal effort for incorporating coherence errors detection into a system. The cache controllers have to be extended to add the required bits to the coherence messages, and the checkers have to be connected to the network.

Compared to [129] (see Section 2.6), our scheme requires significantly less hardware and performance overhead. In [129], every state change requires the controller to submit the new state and some other related information to the checker. We only require the controller to add a few bits to the broadcast messages. Furthermore, the global verification in [129] requires a dedicated logical network. This network can use the existing hardware communication resources, increasing the network traffic and potentially introducing performance overhead. Alternatively, this network can use additional, dedicated hardware resources, increasing the cost. Our approach does not feature a global verification. Consequently, no hardware and/or time is required for that. In addition, unlike [129], our approach not only verifies the state transitions of cache lines, but also makes sure that appropriate messages appear on the network.

Token Coherence Signature Checker (TCSC) [130] (see Section 2.6) can be considered more general than our scheme, since it not only targets snooping-based coherence protocols. As the proposal described in [129], TCSC performs global conflicts checking, while we do it only to a limited extent. Our scheme, however, introduces almost no network bandwidth overhead, which means a minimal cost and no performance overhead. TCSC is scalable in the sense that the hardware overhead is almost linear in the number of caches. However, TCSC requires verifiers, which can become a bottleneck in systems with a large number of caches. In this case, TCSC adds additional verifiers, possibly creating hierarchies of them. The hardware overhead of our approach is always linear in the number of caches. In our scheme, however, like in [129], the hardware overhead per cache depends on the size of the cache, because the state of the cache lines is kept in the checkers. In TCSC the overhead per cache does not depend on the size of the cache.

Compared to the work of Sorin, Hill, and Wood [132] (see Section 2.6), the technique proposed in this chapter requires less bandwidth overhead at the expense of a weaker interconnect fault coverage. Unlike the work of Fernandez-Pascual et al. [134], which targets faults in the interconnection network, this chapter mostly targets faults in the cache coherence logic.

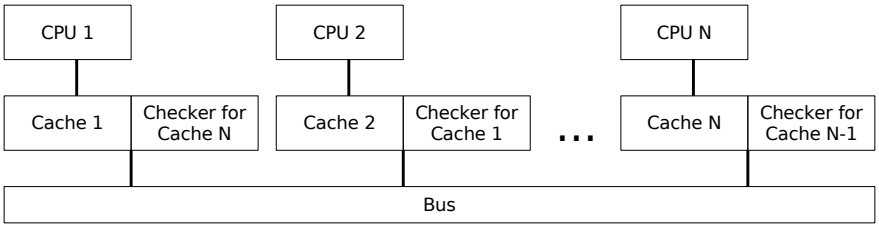


Figure 6.1: Proposed system structure.

6.3 Cache Coherence Verification

This section presents a design implementing the proposed technique. General design decisions, which are suitable for many snooping-based coherence protocols, and their motivation are described in Section 6.3.1. A detailed implementation example, specific for the widely used MESI coherence protocol, is discussed in Section 6.3.2.

6.3.1 General Design Decisions

The cache coherence verification technique proposed in this chapter addresses multiprocessor systems with a number of nodes, each with a private cache, connected with a network, such as a bus. Coherence is achieved by means of a snooping-based protocol [125], such as MESI [150]. The checkers can reside on other than the controlled caches, or on separate bodies connected to the network. The checkers should preferably not reside on the controlled caches, because this increases the coverage of network faults (for example, network messages not received by the cache might be received by the remote checker). If the checkers reside on other caches, some logic processing the incoming messages can be shared between the checker and its host cache, reducing the hardware overhead introduced by the checkers. Thus, such a design is preferred. Figure 6.1 depicts the proposed system structure with a bus as the interconnection network. In Figure 6.1, cache number *i* incorporates a checker for the cache number *i-1*. This is not necessary, however, any distribution of the checkers is possible. Later it will be shown that placing checkers as far as possible from the controlled caches is even desirable for increasing the network fault coverage.

A checker needs some information about each line of the controlled cache,

such as its block address and current coherence protocol state. Hence, the checker keeps the tag of each line, but not the data. Keeping the data provides no benefit. It implies a large hardware overhead, but does not help error detection. Verifying the data would require every local processor write to be broadcast on the network (to inform the checker), which implies a network traffic overhead that we do not consider feasible. It is more practical to apply conventional protection methods to the data, such as error detection/correction codes [9].

The controlled cache could notify the checker about all state transformations through a special connection between them. However, this would require special logic added to the cache, as well as a dedicated communication line between the cache and its checker. These communication lines would complicate the final design, requiring to place and route them, and prevent locating checkers at arbitrary places. Furthermore, adding extra hardware not only increases the system cost, but also introduces new potential faulty points. To minimize the hardware and performance overhead, we aim at making the checkers as autonomous as possible. Any special communication between the caches and the checkers is avoided. The checker only snoops the network, processing all the messages related to the data in the controlled cache. Based on this information, it maintains its own states, and verifies the states on the controlled cache. A mismatch of the checker's and the controlled cache's states for a cache line signals an error. In addition to this, by snooping the messages from other caches, the checker makes sure that the states in other caches do not conflict with the states on the controlled cache. Besides the states, the checkers verify the network transactions, initiated both by the controlled cache and the rest of the system (addressed to the controlled cache). For example, it makes sure that every memory request of the controlled cache is answered, that the controlled cache does not attempt to write back an invalid cache line, etc. Thus, by connecting the checkers to the system network instead of a dedicated communication line, we gain some additional control of the network interface (both of the checked cache and of other entities communicating with it), and the network itself.

The effectiveness of the proposed checker design can be compared to the traditional logic duplication with comparison of the results. Inside the cache, the logic implementing coherence, the coherence protocol state and tags could be duplicated. Every state transition would require an agreement between the replicated logic. The independent checker has several advantages over this scheme. An independent checker is most probably physically located further away on the die (or on a different die) from the checked logic. This reduces the probability of common faults. A duplicated logic without modifications

inherits possible design faults. The checker should be designed independently, thus it has the potential to detect design and implementation faults. Furthermore, as discussed earlier, the checker can implement additional functionality, such as verification of the network operation. The price to be paid for these advantages is a possible increase of hardware cost (if the checker needs more logic than the duplicated version). If the design requires additional network traffic, a certain performance overhead is also likely to appear.

The proposed technique is able to detect errors in the cache coherence logic and in the storage elements where the current states are kept. Upon error detection, depending on the design, the system can be halted, a graceful degradation may take place, or a recovery may be initiated. In the case of graceful degradation, the erroneous node can be switched off in a multiprocessor environment, and its task is reassigned to a different processor. Optionally, assuming the fault could be transient, the node can be switched on again and tested for permanent faults. If the system supports a recovery mechanism, such as [151, 152], it can restart operation from the last state which is known to be correct. The technique presented in this chapter does not favor recovery, because the latency between when the error occurs and when it is detected is not fixed. Errors are detected when a message holding the erroneous state appears on the network. Checkpointing requires that at some moment the system state is known to be correct. Additional activities are required for this in the proposed scheme. For example, the caches can send the current states of all the lines to the checkers for verification. If the verification result is positive, a checkpoint can be created.

The proposed technique relies on the correctness of the system network. Faults leading to network messages loss, for example, are not reliably covered. If a request from one cache neither reaches another cache nor its checker, the error is not going to be detected, unless it will lead to further conflicting coherence actions. This is another reason why it is better to place checkers further away from the checked caches in the network: the chance that a lost message will reach at least one of them increases.

6.3.2 Protocol-Specific Implementation

In this section the proposed idea is applied to the widely used MESI cache coherence protocol [150]. The MESI protocol features four possible states for a cache line: *Modified* (*M*), *Exclusive* (*E*), *Shared* (*S*), and *Invalid* (*I*). In the *M* state, a valid copy of the line is present only in the owner's cache. The local processor has modified the line and can write it again without notifying the

other nodes, so it should be invalidated in all the other caches. The *E* state means that both the memory and the cache have a valid copy, but other caches do not. The local processor can write it, switching its state to *M*, without notifying other caches. The *S* state indicates that the line is present in multiple caches and in memory. The caches use it in read-only mode. A write access requires sending a notification (invalidation) to other nodes. The *I* state signals that the data in the line are invalid.

The implementation of the MESI protocol which is discussed in this work, with the required state transitions in response to different requests, is shown in Table 6.1. The first column contains the request, coming either from the local processor or from the network. The following requests and network messages appear:

- *PrRd* – read request from the local processor.
- *PrWr* – write request from the local processor.
- *BusRd* – read request snooped from the network.
- *BusRdX* – read exclusive request from the network.
- *BusWB* – write back from a cache.
- *Flush* – flush request snooped from the network.

The second column defines the current state at which the request is received. The third column shows the resulting state of the transition induced by the request. The fourth column lists messages that the cache sends to the network in response to the received request. The fifth column defines the corresponding actions required from the coherence checker of this cache. The following abbreviations are used:

- == *state* : make sure the current state is *state* (else report an error).
- $\rightarrow$ *state* : change the state to *state*.
- *Answer* : make sure the request is answered.
- *No Answer* : make sure request is not answered.
- *Signal Error* : report an error and stop operation.

Table 6.1: MESI protocol state transitions and corresponding actions of the cache and its checker.

Request	Current State	Next State	Bus Message	Checker Activity
PrRd	*M*	*M*	-	-
PrRd	*E*	*E*	-	-
PrRd	*S*	*S*	-	-
PrRd	*I*	*E/S*	*BusRd*	$== I$; *Answer*; $\rightarrow$ *E/S*
PrWr	*M*	*M*	-	-
PrWr	*E*	*M*	-	-
PrWr	*S*	*M*	*Flush*	$== S$; $\rightarrow M$
PrWr	*I*	*M*	*BusRdX*	$== I$; *Answer*; $\rightarrow M$
BusRd	*M*	*S*	*BusWB*	$\rightarrow S$; *Answer*
BusRd	*E*	*S*	*BusWB*	$\rightarrow S$; *Answer*
BusRd	*S*	*S*	*BusWB*	*Answer*
BusRd	*I*	*I*	-	*No Answer*
BusRdX	*M*	*I*	*BusWB*	$\rightarrow I$; *Answer*
BusRdX	*E*	*I*	*BusWB*	$\rightarrow I$; *Answer*
BusRdX	*S*	*I*	*BusWB*	$\rightarrow I$; *Answer*
BusRdX	*I*	*I*	-	*No Answer*
Flush	*M*	-	-	*Signal Error*
Flush	*E*	-	-	*Signal Error*
Flush	*S*	*I*	-	$\rightarrow I$; *No Answer*
Flush	*I*	*I*	-	*No Answer*

Local processor read requests (*PrRd*) for cache lines in any state except *I* are serviced within the node and do not induce any network activity. They also do not change the states, thus the checker does not need to be notified about these events. *PrRd* for an invalid line leads to a *BusRd* requesting the line from the memory. Thus, when the checker snoops a *BusRd* request from the checked cache, it knows that this request can only appear for an invalid cache line, and checks if the corresponding line is in the *I* state in its cache. Further, the checker makes sure that the request is answered. The architecture

discussed here uses the conventional approach in which other caches answer read requests if possible. If other caches do not contain the requested data, the memory answers. If the *BusRd* request is answered by the memory, the cache line is not present in other caches, so its state becomes *E*. Otherwise, other caches share the line, so the state becomes *S*.

Local processor writes (*PrWr*) to lines in the state *M* do not alter the state and do not produce any bus messages, thus the checker does not perform any action. *PrWr* to a line in the state *E* is the only event which changes the state (to *M*), but which is not associated with any network traffic. The checker can, therefore, not know about this transition without a special notification from the checked cache. Thus, a design decision has to be made in this case. The checked cache could broadcast a special notification message for the checker. However, this implies that the cache controller has to be redesigned, and some additional network traffic appears, which could possibly degrade performance if the network throughput is an issue. There exists another solution, acceptable in this case. The *E* to *M* transition is considered safe, and it is not communicated to the checker. This is reasonable because an erroneous *E* to *M* transition will never affect the data integrity in the system, as can be seen from Table 6.1. An illegal *S* to *M* transition, for example, would lead to an absence of the necessary invalidation (*Flush*) request on future *PrWr*. This means that copies on other caches would not be invalidated, and other processors would read wrong (stale) data. However, a similar problem propagating to the rest of the system cannot happen on an illegal *E* to *M* transition. Thus, to minimize the performance and cost overhead, this transition is allowed. If, after a silent *E* to *M* transition in a cache, its checker receives a line in state *M* instead of the expected *E*, it does not signal an error, but changes the state to *M* to reestablish coherence with the checked cache. If an external request arrives for a line whose *M* and *E* states are not synchronized between the cache and the checker yet, it is not problematic. As can be seen from Table 6.1, the checker actions in response to external requests *BusRd*, *BusRdX* and *Flush* are the same for the states *M* and *E*. For the external read requests, the coherence states are changed to the same one (*S* in the case of *BusRd* request and *I* in the case of *BusRdX* request), thus the cache and the checker become coherent again. Hence, no error will be reported or propagated. *PrWr* on a line with the *S* state changes the state to *M* and broadcasts an invalidation (*Flush*) request on the bus. Hence, when a checker snoops a *Flush* request from the checked cache, it makes sure that the old state is *S* (there are no other states that produce the *Flush* request), and updates the state. When a local write miss happens, the cache sends the *BusRdX* request, to receive the necessary line and invalidate it in other caches.

In this case the checker makes sure the current state is *I*, that an answer arrives, and changes the state to *M*.

When the checker snoops a read request (*BusRd*) for a line which is present and valid in the checked cache, it changes the state to *S* if necessary, and makes sure that the checked cache answers with a *BusWB* message. If the requested line is invalid in the checked cache, the checker makes sure it does not answer. On a *BusRdX* request for a valid line, the checker invalidates the line, and makes sure the checked cache sends a *BusWB* answer. *BusRdX* for an invalid line should produce no answer from the checked cache.

Upon receiving a *Flush* request from the network, the checker invalidates the corresponding line if it is in the state *S*, and makes sure the checked cache does not answer. If a *Flush* request is received for a line in the *M* or *E* state, an error is reported, because it is an illegal situation: only lines in the *S* state can send a *Flush* request. The checked cache cannot have a line in the *M* or *E* state if it is present in the *S* state in another cache. Thus, this situation signals an error.

Table 6.1 demonstrates the checker's actions corresponding to MESI transitions in the checked cache. It only partially specifies the actual checker implementation, and does not reflect some details like cache line evictions induced by local misses. A full event-driven implementation specification is given in Table 6.2. The checker snoops a network message, and if it relates to data contained in the checked cache, classifies it as a checked cache request, answer for the checked cache, or request from another cache. It is further processed depending on the particular message, as shown in Table 6.2. The following additional actions complete the specification of the checker:

- for all messages (requests and answers) snooped from the checked cache, the checker verifies the current state of the line, if available. This is the action which actually detects illegal state transitions and reports errors.
- the checker makes sure that every request from other caches, if it hits, is answered by the checked cache.
- the checker makes sure that the checked cache never answers with invalid data (lines with the state *I*).
- the checker makes sure that every answer to the checked cache comes in response to a corresponding request.

There are several necessary design decisions not mentioned so far. To be able

Table 6.2: Checker actions in response to snooped bus traffic.

Bus Message	**Checker Activity**
Requests from the checked cache:	
BusRd or *BusRdX*	== *I*: the line is either already invalid, or has previously been evicted and invalidated. *Answer*.
BusWB	→ *I*: this should be an eviction. If another request follows, make sure it is *BusRd* or *BusRdX*.
Flush	== *S*; → *M*.
Answers for the checked cache:	
BusRd answer from memory	→ *E*.
BusRdX answer from memory	→ *M*.
BusWB as an answer from another cache	If *BusRd* was requested, → *S*. If *BusRdX* was requested, → *M*.
Requests from other caches:	
BusRd	If state is I: *No Answer*. If state is not I: → *S*; *Answer*.
BusRdX	If state is I: *No Answer*. If state is not I: → *I*; *Answer*.
Flush	== *S* or *I*; → *I*; *No Answer*

to check the current state of a line in the checked cache, the checker needs to receive this state. This could be achieved by letting the cache send notifications on every state transition to its checker. However, as discussed earlier, for optimization reasons this is avoided. In this work the problem is solved by extending the network message with the current state of the addressed cache line. All messages sent by the cache include the current state information of the addressed line, which is used by the checker.

Another problem appears if the cache is not direct-mapped, but features associativity. Upon a miss, the checker needs to know where the incoming data will be placed, to maintain the correct information about the lines in the checked cache. The set can be easily determined by the address which is present in the request. However, depending on the replacement policy used, the way number is determined randomly or based on the access history. For line re-

placement policies such as Least Recently Used (LRU), the checker needs the whole cache access history to determine the way which will be used. It is not feasible to send all this information from the cache, thus another solution is needed. In this work it is solved by further extending the network message with the way number of the cache line.

Thus, to support the proposed technique, the network message format is extended with 2 bits for the current MESI state, and a certain number of bits for the way where a cache line resides (depends on the cache associativity). For example, 1 bit is sufficient for a 2-way set-associative cache. Hence, a total of 3 additional bits are added to the message for the MESI protocol and 2-way set-associative caches.

6.4 Fault Coverage

This section discusses the fault coverage of the proposed coherence verification method. The MESI-specific design presented in Section 6.3.2 is considered. Section 6.4.1 analytically evaluates the fault coverage, and Section 6.4.2 presents experimental results.

6.4.1 Analytical Fault Coverage Evaluation

Assuming an error-free communication network, the proposed method detects all coherence errors visible to the checker. Note that by coherence errors we mean illegal changes of the coherence protocol states in cache lines. These errors can be due to faults in the new state generation logic, or faults in the storage elements holding the current states. In addition, some communication errors such as the absence of answers to issued requests are also covered.

A coherence state error is visible to the checker when the controlled cache broadcasts any message related to the corresponding cache line, either a request or a reply. The message contains the current coherence state of the line, which is verified by the checker. In addition, the absence of an expected reply to requests from other nodes in the system can signal errors. For example, an illegal transition of a valid line to the *I* state will lead to the absence of a reply to a read request. Thus, coherence state errors are not detected in the lines which, after the errors appear, never send and receive requests, and are never written back to the memory.

Consider a cache line which, after its coherence state becomes erroneous,

is never addressed by requests from other nodes in the system and is never evicted. When the node's task execution completes, this line will be written back only if it has been changed by the local CPU, i.e. if its current state is *M*. The lines in other states do not have to be written back, because they have a valid copy in the memory. In this situation, the erroneous *M* state will be detected by the checker, when the line is written back. Illegal transitions from the state *M* to another state will also be detected, because it does not write back, as expected. Illegal transitions from any state to *I* will also be detected, if the local processor tries to access the line, because the corresponding read request will be issued on the network. If the local processor does not access the line anymore, it has no influence on the data integrity, and the error can be safely ignored. Only the illegal transitions from a non-*M* state to the states *E* and *S* are not covered so far. Their possible combinations and the probable consequences are listed below:

- **Illegal transition from *E* to *S***. This transition is harmless, because the local processor still reads valid data from the line, and does not write to this line.
- **Illegal transition from *S* to *E***. This transition is also harmless. The local processor still reads valid data from the line. If the line is written, however, its state will silently be changed to *M*. Then, the line in the state *M* will eventually try to write back, and the checker will detect the error.
- **Illegal transition from *I* to *E* or *S***. These transitions are the most dangerous for the proposed method. They can lead to local reads receiving garbage. On the base of this garbage other (correct) cache lines can be updated, and the data integrity will be violated. This case will not be detected if the line is never evicted or referenced again from outside. Experimental results in Section 6.4.2 demonstrate that this rare situation is not likely to appear in practice.

To summarize, the proposed method covers all harmful single errors in cache coherence states, except illegal transitions from state *I* to state *E* or *S* in cache lines which are never referenced from the memory system and are never evicted later. Many multiple errors are also covered, except those that assign back the correct state to the cache line before it is checked.

The undetectable single error cases described above are due to the fact that the cache never broadcasts any messages related to the erroneous lines. This

problem could be solved by forcing a broadcast of the line states at a certain time. Because the local CPU does not know which lines are in erroneous states, it has to broadcast the states of all lines. The checker then verifies all the lines. This operation introduces a network bandwidth overhead, and should be performed as rarely as possible. An error may propagate from a node to the system only when the node updates the shared memory, i.e., when it writes back. Thus, a possible solution is to broadcast the states of all the cache lines at every cache write back.

6.4.2 Experimental Fault Coverage Evaluation

To evaluate the fault coverage of the proposed method experimentally and to validate its correctness, a cycle-accurate multiprocessor simulator based on the UNISIM environment [153, 154] has been used. The original simulator available in the UNISIM public repository has been extended to support the proposed technique and to inject faults into the cache coherence states. Simulations have been conducted on a shared memory CMP with two PowerPC 405 32-bit RISC cores. Every CPU has a private data cache, connected to a bus. The cache contains 128 lines, each of size 32 bytes. The cache is 2-way set-associative. A DRAM memory is also connected to the bus. The simulator implements the MESI protocol to maintain cache coherence.

The checkers implementing the logic described in Section 6.3.2 have been designed and integrated into the simulated multiprocessor as shown in Figure 6.1. A fault injector has been implemented which changes the current coherence state of a cache line to a different state, chosen randomly. This simulates both the faults of the cache controller logic which generates the new MESI state based on a certain transaction, and faults in the storage elements holding the current state. The injection time and place is controlled by a random number generator. The injection frequency is controlled by a user-defined injection period. A synthetic benchmark has been used in which the different nodes perform many read and write accesses to shared memory locations. This ensures that many coherence actions take place. The output produced in the presence of faults has been compared to the correct output to determine if the faults have affected the final result, which is of most importance for the end user. The fault injection frequency was varied from extremely high (roughly one fault per 100 clock cycles) to low (one fault per simulation).

14380 simulations with one or more injected faults have been performed. 6290 times the checker detected the error(s), 2466 times the simulator detected the error(s), and 5624 times the simulation finished with undetected faults, but the

faults did not propagate to the output and did not corrupt it (these are escapes). We see the following possible causes of the escapes: the faults were injected into cache lines which were never referenced again in such a way that wrong data propagated in the system (see Section 6.4.1), or the faults were masked, i.e. the following fault(s) returned the correct state, or the state was changed by future coherence transactions. None of the simulations finished with undetected faults and corrupted output. The theoretically possible situations in which undetected faults violate the system data integrity and affect the output (see Section 6.4.1) never occurred in our simulations. This provides additional evidence that they are, indeed, very rare.

The storage overhead needed for the tags and coherence states of each checked cache line in the checker for the simulated cache configuration can be calculated as follows. The 32-bit address consists of a 21-bit tag, a 6-bit index, and a 5-bit block offset. Only the 21-bit tag and 2-bit MESI state need to be saved on the checker for each cache line. Since every line stores 32 bytes of data, the overhead is $\frac{21+2}{21+2+32\cdot 8} = 0.08$, which is 8%.

6.5 Conclusions

This chapter has presented a technique to dynamically verify the cache coherence protocol operation on snooping-based multiprocessor systems. The widely used MESI coherence protocol has been analyzed in detail. A careful design and a limited speculation (such as accepting illegal E to M state transitions which are considered safe for system data integrity) allowed to completely avoid any performance overhead for the MESI protocol. This indicates that applying a similar approach to other protocols can also be done with limited or no performance overhead. The hardware cost is constant per processing node: for every cache, a fixed amount of redundant logic (the checker) is required. Thus, the technique is scalable. In addition, every checker contains copies of all the tags of all cache lines in the checked cache, which introduces approximately 8% storage overhead for the simulated architecture. Tags overhead depends on the cache size, but not on the number of nodes in the system.

Only a few simple changes to an existing system are required to support the technique. The checkers can be easily integrated by connecting them to the communication network so that they can snoop all the messages. The message has to be extended with a few additional bits (3 in case of the MESI protocol and 2-way set-associative caches). The cache controllers need a simple modification to integrate the information about the way number and the current

coherence state into every message they send.

The technique addresses faults in the logic controlling the coherence transactions, and in the storage elements holding cache line states. Almost all single-fault scenarios are covered, except for a few rare cases described in Section 6.4.1. Experimental results show that these cases are very rare: none of over 14 thousand simulations finished with undetected faults leading to corrupted output. Furthermore, the technique can be strengthened by using multiple checkers per cache if very high error rates are expected.

To provide a full system coverage, the proposed technique is assumed to be used in combination with other detection and/or correction techniques. These techniques would cover the faults within the nodes, and the network faults which are only partly covered by the proposed scheme. Note that while the logic (except the cache coherence controllers) and memory on all the nodes need extra protection, the coherence checkers proposed in this work do not necessarily need to be error-free. This is because the cache controllers are essentially duplicates of the checkers, thus they check the operation of each other.

The technique presented in this work aims at a fail-safe operation. To enable recovery, some modifications are required, because in the fail-safe mode error detection latency is unpredictable. For example, to implement checkpointing, the caches can submit the states of all the lines to the checkers for verification. If no errors are detected, a checkpoint can be made. This, however, is likely to introduce performance overhead. Extensions of the proposed technique to support recovery as well as applications to other than MESI coherence protocols are planned for the future work.

It might be impossible to avoid additional network messages to apply the proposed scheme to other than MESI cache coherence protocols, or to enable fault recovery. In this case, to avoid performance penalty, it might be useful to investigate if the additional network messages can be issued only when the network is free from other messages.

Note. This chapter is based on the following paper:

Demid Borodin and B.H.H. (Ben) Juurlink, **A Low-Cost Cache Coherence Verification Method for Snooping Systems**, *DSD'2008: Proceedings of the 11th Euromicro Conference on Digital System Design*, pp. 219–227, September 2008.

7 Conclusions

This dissertation has addressed the reduction of the performance (and in many cases energy consumption) penalty introduced by time redundant FT techniques. Time redundant FT techniques are often preferred to space redundant methods when designing low-cost non-critical systems, such as general purpose computers and embedded systems, because the traditional space redundancy methods, such as replication of the whole system to verify its results, are too expensive for the considered markets.

This dissertation has proposed several such Performance-Oriented Fault Tolerance (POFT) techniques. These are both specific low-cost FT techniques and universal approaches that can be applied to many different time redundant FT techniques to reduce their performance (and in some cases energy consumption) penalty. Moreover, these are both approaches that can be used with different system architectures, and techniques targeting particular architectures, such as snooping multiprocessor systems supporting cache coherence.

This chapter summarizes the work presented in the dissertation. Section 7.1 summarizes the dissertation. Section 7.2 presents the major contributions. Finally, Section 7.3 proposes future research directions.

7.1 Summary

The current technology trends leading to the need of FT features even in low-cost non-critical computing systems have been introduced in Chapter 1. The types of protective redundancy and their associated overhead have been discussed.

Chapter 2 has presented the background information to introduce the reader to the subject. The history of FT development has been briefly introduced.

Then, the conventional FT techniques have been described, that function as a base for the majority of the existing techniques. After that, some previously proposed FT techniques for different computer architectures reducing performance overhead have been discussed. Special attention has been paid to FT techniques implemented in software, because they are useful for the targeted low-cost systems. Finally, the focus has been placed on the FT issues specific for the currently very popular multiprocessor systems.

Chapter 3 has presented a novel approach to FT proposed in this dissertation called Instruction-Level Configurability of Fault Tolerance (ILCOFT). ILCOFT allows an application developer to assign different protection levels to various application parts. Chapter 3 has explained why this is useful and has discussed possible implementation strategies. Then the possible ways to apply ILCOFT to various existing FT techniques have been discussed. Finally, this approach has been evaluated from the performance, energy consumption, and fault coverage points of view.

The most challenging requirement when using ILCOFT is to assign the necessary protection levels to different application parts (instructions or blocks of instructions). In other words, to decide how critical every instruction is. When performed manually by the application developer, this task requires a large effort and is very error-prone. It can be done automatically by the compiler, but this approach is based on the unsafe assumption that only instructions affecting the application control flow are critical. Data processing instructions that may often corrupt the whole application output are never protected in this way. Chapter 4 has proposed another method to evaluate how critical individual instructions are. The notion of Instruction Vulnerability Factor (IVF), analogous to the Architecture Vulnerability Factor (AVF), but addressing instructions instead of architectural units vulnerability, has been introduced. A methodology to estimate the IVF has been proposed in Chapter 4. Then IVF has been used by the ILCOFT-enabled instruction duplication fault detection scheme. Instructions with high IVF have been duplicated, while others have not. Experimental results have demonstrated that this method achieves a significant performance improvement, while preserving a good fault coverage.

Chapter 5 has proposed to use instruction precomputation for fault detection purpose. Instruction precomputation has been used to supplement the instruction duplication fault detection technique. Experimental results have demonstrated that this configuration achieves better performance and fault coverage than instruction duplication alone. It has been shown that precomputation improves the long-lasting transient and permanent fault coverage of the existing

scheme which uses instruction memoization in a similar manner. In addition, Chapter 5 has proposed to combine instruction precomputation and memoization. It has been shown that the combination achieves better performance than any one of these techniques alone does. The combination used with instruction duplication has also been shown to outperform precomputation or memoization used with instruction duplication alone.

Finally, Chapter 6 has presented an error detection method to verify the cache coherence operation in snooping multiprocessor systems. First, a general design suitable for different snooping-based systems has been presented. Then it has been applied to systems utilizing the MESI cache coherence protocol. Experimental results have shown that the proposed method verifies the MESI coherence protocol operation without any performance overhead. To support the proposed method, additional checkers have to be connected to the system network, and a few simple modifications are required in the cache controllers. This is achieved at the expense of a slight fault coverage limitation, which has never manifested itself in the experimental results.

7.2 Contributions

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

- **A novel approach to FT called ILCOFT has been proposed.** ILCOFT enables an application developer to protect critical application parts better than non-critical ones. This allows to trade reliability with the overhead (performance penalty and energy consumption increase) reduction. Moreover, if more critical application parts are protected better at the expense of the weaker protection of the less critical parts, the overall reliability can be improved without introducing any additional overhead.

- **The notion of Instruction Vulnerability Factor (IVF) has been introduced.** IVF determines how much of the final application output is corrupted due to faults in every instruction. IVF serves to assign appropriate protection levels when using ILCOFT. IVF releases an application developer from the need to assign protection levels manually, which requires a significant effort and is very error-prone. Unlike the automatic protection level assignment performed by the compiler, IVF does not assume that only instructions affecting the application control flow are

critical, and thus it is more accurate and suitable for a larger range of applications.

- **Instruction precomputation has been proposed to supplement the instruction duplication error detection technique.** Instruction precomputation improves the performance and fault coverage of instruction duplication.
- **The combination of instruction precomputation and memoization has been proposed.** The combination of these result reuse techniques is shown to outperform every one of them used alone. The combination of these techniques used in conjunction with the instruction duplication error detection technique outperforms and increases the fault coverage of precomputation or memoization used with instruction duplication alone.
- **A low-cost cache coherence verification method has been proposed.** This method has been shown to have no performance penalty in the case of the MESI cache coherence protocol. Only a few minor hardware modifications are required in the system to support the coherence verification. This is achieved at the expense of a slightly reduced fault coverage (compared to other more expensive methods). This fault coverage limitation has never manifested itself in the experimental results.

7.3 Possible Future Directions

The author advocates further research on FT overhead reduction. New techniques, possibly based on existing performance improvement methods such as instruction precomputation and memoization, can be expected to appear.

The author especially recommends the philosophy that not everything needs to operate perfectly to achieve the desired goal. The ILCOFT approach proposed in this dissertation is based on this philosophy, allowing some (non-critical) instructions to be unprotected. This achieves a better overall performance, cost, and/or reliability. A similar approach can be investigated, for example, for different hardware units, protecting some of them better than others. Alternatively, the desired protection level can be determined based on the usage characteristics of hardware units. For example, floating-point FUs are used rarely compared to integer FUs, thus, their protection can be weaker.

The possible future research directions to improve the techniques proposed in this dissertation are the following. More metrics (such as IVF) of how criti-

cal every application instruction is are desirable to be used by ILCOFT. It is preferable to invent automatic methods to estimate these measures. This would eliminate the necessity of manual work, which is error-prone. Different protection level assignment methods can target different objectives. For example, if it is only important that an application does not crash, the automatic compiler assignment proposed in Chapter 3 is most appropriate, because it will reduce maximum overhead. If more accurate results are necessary, IVF is a more suitable measure (see Chapter 4).

The IVF estimation method presented in this dissertation (Chapter 4) can be further improved. First of all, it is desirable to focus on improving the speed of the IVF estimation process. Furthermore, some enhancements of our experimental setup are desirable (the details are discussed in Section 4.4). For example, a timer can be used to detect cases when faults in instructions lead to significant performance penalties. The following approach can be used: all the instructions affecting the application control flow should be assigned the maximum IVF by the compiler, and the other instructions should be evaluated using profiling. This would both speed up the IVF estimation process, and solve the problem of faults significantly increasing the application execution time. In addition, it is desirable to adapt the proposed IVF estimation method for applications producing output consisting of parts of varying importance. For example, in a video sequence, bytes defining individual pixel values are less important than bytes defining frame attributes. The adapted IVF estimation method should be able to assign different weight to the corruption of more and less important output elements.

The techniques involving instruction precomputation and memoization (Chapter 5) can be investigated to further optimize the proposed system configurations. For example, in the combined scheme in this work, precomputation and memoization used equally sized tables. It would be interesting to investigate how memo- and P-tables of different sizes influence the performance and fault coverage. Moreover, it is possible to reduce the number of instructions appearing in both the P- and memo-tables by using trivial computation detection, which can be expected to further improve performance.

The cache coherence verification technique proposed in Chapter 6 currently supports only error detection. Extension of this technique with error recovery capabilities appears to be difficult. It requires certain non-trivial modifications that can be expected to degrade the performance, as briefly discussed in Section 6.5. It would be interesting to investigate if such an extension is feasible. In addition, in the case if introducing additional network messages for cache

coherence verification/recovery is unavoidable, the performance penalty it incurs should be minimized. This can be achieved, for example, by making sure that the additional messages are only issued to the system network when it has no other messages, or when additional messages will not degrade its performance.

Bibliography

[1] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic Concepts and Taxonomy of Dependable and Secure Computing," *IEEE Transactions Dependable Secur. Comput.*, vol. 1, no. 1, pp. 11–33, 2004.

[2] A. Avizienis, "Fault-Tolerance and Fault-Intolerance: Complementary Approaches to Reliable Computing," in *Proc. Int. Conf. on Reliable software*. New York, NY, USA: ACM Press, 1975, pp. 458–464.

[3] B. Johnson, *Design and Analysis of Fault-Tolerant Digital Systems*. Addison-Wesley, Jan 1989.

[4] P. I. Rubinfeld, "Managing Problems at High Speed," *IEEE Computer*, vol. 31, no. 1, pp. 47–48, 1998.

[5] J. F. Ziegler, "Terrestrial cosmic rays," *IBM Journal of Research and Development*, vol. 40, no. 1, pp. 19–39, 1996.

[6] T. May and M. Woods, "Alpha-Particle-Induced Soft Errors in Dynamic Memories," *IEEE Transactions on Electron Devices*, vol. 26, no. 1, pp. 2–9, 1979.

[7] A. Avizienis, "Dependable Computing Depends on Structured Fault Tolerance," in *Proc. 6th Int. Symp. on Software Reliability Engineering*, 1995, pp. 158–168.

[8] ——, "Fault Tolerance: the Survival Attribute of Digital Systems," *Proc. IEEE*, vol. 66, no. 10, pp. 1109–1125, Oct 1978.

[9] T. Rao and E. Fujiwara, *Error-Control Coding for Computer Systems*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1989.

[10] A. Avizienis, "Fault-Tolerant Computing: An Overview," *IEEE Computer*, vol. 4, no. 1, pp. 5–8, Jan 1971.

[11] ——, "Framework for a Taxonomy of Fault-Tolerance Attributes in Computer Systems," in *ISCA-83: Proc. 10th Annual Int. Symp. on Computer architecture*. Los Alamitos, CA, USA: IEEE Computer Society, 1983, pp. 16–21.

[12] D. A. Rennels, "Fault-Tolerant Computing – Concepts and Examples," *IEEE Transactions on Computers*, vol. C-33, no. 12, pp. 1116–1129, Dec 1984.

[13] A. Avizienis, "Dependable computing: From concepts to design diversity," in *Proc. IEEE*, vol. 74, no. 5, May 1986, pp. 629–638.

[14] ——, "The Dependability Problem: Introduction and Verification of Fault Tolerance for a Very Complex System," in *ACM-87: Proc. Fall Joint Computer Conf. on Exploring technology: today and tomorrow*. Los Alamitos, CA, USA: IEEE Computer Society, 1987, pp. 89–93.

[15] D. P. Siewiorek, "Fault Tolerance in Commercial Computers," *IEEE Computer*, vol. 23, no. 7, pp. 26–37, 1990.

[16] A. Avizienis, "Toward Systematic Design of Fault-Tolerant Systems," *IEEE Computer*, vol. 30, no. 4, pp. 51–58, 1997.

[17] A. K. Somani and N. H. Vaidya, "Understanding Fault Tolerance and Reliability," *IEEE Computer*, vol. 30, no. 4, pp. 45–50, Apr 1997.

[18] A. Avizienis and Y. He, "Microprocessor Entomology: A Taxonomy of Design Faults in COTS Microprocessors," *Dependable Computing for Critical Applications 7*, pp. 3–23, Jan 1999.

[19] J. Yi, R. Sendag, and D. Lilja, "Increasing Instruction-Level Parallelism with Instruction Precomputation," in *Euro-Par-02: Proc. 8th Int. Euro-Par Conf. on Parallel Processing*. London, UK: Springer-Verlag, Aug 2002, pp. 481–485.

[20] D. Michie, "Memo Functions and Machine Learning," *Nature 218*, pp. 19–22, 1968.

[21] S. Richardson, "Exploiting Trivial and Redundant Computation," in *Proc. 11th Symp. on Computer Arithmetic*, July 1993, pp. 220–227.

[22] A. Sodani and G. S. Sohi, "Dynamic Instruction Reuse," in *ISCA-97: Proc. 24th Annual Int. Symp. on Computer Architecture*. New York, NY, USA: ACM, 1997, pp. 194–205.

[23] M.N.O.Sadiku and C.N.Obiozor, "Evolution of Computer Systems," in *Proc. of Frontiers in Education Conf., 1996. FIE '96. 26th Annual Conf.*, vol. 3, Salt Lake City, UT, Nov 1996, pp. 1472–1474.

[24] W. Carter and W. Bouricius, "A Survey of Fault Tolerant Computer Architecture and its Evaluation," *IEEE Computer*, vol. 4, no. 1, pp. 9–16, Jan 1971.

[25] J. Brainerd and T. Sharpless, "The ENIAC," in *Proc. IEEE*, vol. 72, no. 9, Sep 1984, pp. 1203–1212.

[26] G. Candea, "The Basics of Dependability," Sep 2003, lecture notes, available at http://swig.stanford.edu/~candea/teaching/cs444a-fall-2003/notes/basics.pdf (last accessed 18.09.2006).

[27] W. Fritz, "ENIAC - A Problem Solver," *IEEE Annals History of Computing*, vol. 16, no. 1, pp. 25–45, 1994.

[28] J. Eckert, J. Weiner, H. Welsh, and H. Mitchell, "The UNIVAC system," *AIEE-IRE Conf*, pp. 6–16, Dec 1951.

[29] M. Hsiao, W. Carter, J. Thomas, and W. Stringfellow, "Reliability, Availability, and Serviceability of IBM Computer Systems: A Quarter Century of Progress," *IBM Journal of Research and Development*, vol. 25, no. 5, pp. 453–465, 1981.

[30] D. Siewiorek, "Architecture of Fault-Tolerant Computers: An Historical Perspective," in *Proc. IEEE*, vol. 79, no. 12, Dec 1991, pp. 1710–1734.

[31] J. von Neumann, "Probabilistic Logics and the Synthesis of Reliable Organisms from Unreliable Components," in *Automata Studies*, ser. Annals of Mathematics Studies. Princeton, NJ: Princeton University Press, 1956, vol. 34, pp. 43–98.

[32] J. Lala and R. Harper, "Architectural Principles for Safety-Critical Real-Time Applications," *Proc. of the IEEE*, vol. 82, no. 1, pp. 25–40, Jan 1994.

[33] P. K. Lala, Ed., *Self-Checking and Fault-Tolerant Digital Design*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001.

[34] W. E. Carter, H. C. Montgomery, R. J. Preiss, and H. J. Reinheimer, "Design of Serviceability Features for the IBM System/360," *IBM Journal of Research and Development*, vol. 8, no. 2, pp. 115–126, 1964.

[35] D. Siewiorek and R. Swarz, *Reliable Computer Systems: Design and Evaluation*, 3rd ed. A K Peters Ltd, Oct 1998.

[36] D. Pradhan, *Fault-Tolerant Computer System Design*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., Feb 1996.

[37] S. Hareland, J. Maiz, M. Alavi, K. Mistry, S. Walsta, and C. Dai, "Impact of CMOS Process Scaling and SOI on the Soft Error Rates of Logic Processes," *VLSI Technology. Digest of Technical Papers*, pp. 73–74, 2001.

[38] C. C. Corporation, "Data Integrity for Compaq NonStop Himalaya servers," Apr 1999, white Paper, available at http://www.himalaya.compaq.com/object/dataigwp.html (last accessed 23.10.2006).

[39] C. Webb and J. Liptay, "A High-Frequency Custom CMOS S/390 Microprocessor," *iccd*, vol. 00, p. 241, 1997.

[40] L. Spainhower and T. A. Gregg, "G4: A Fault-Tolerant CMOS Mainframe," in *FTCS-98: Proc. The Twenty-Eighth Annual Int. Symp. on Fault-Tolerant Computing*. Washington, DC, USA: IEEE Computer Society, Jun 1998, pp. 432–440.

[41] L. Spainhower, J. Isenberg, R. Chillarege, and J. Berding, "Design for Fault-Tolerance in System ES/9000 Model 900," in *FTCS-22: The Twenty-Second Int. Symp. on Fault-Tolerant Computing*, Jul 1992, pp. 38–47.

[42] T. Slegel, R. A. III, M. Check, B. Giamei, B. Krumm, C. Krygowski, W. Li, J. Liptay, J. MacDougall, T. McPherson, J. Navarro, E. Schwarz, K. Shum, and C. Webb, "IBM's S/390 G5 Microprocessor Design," *IEEE Micro*, vol. 19, no. 2, pp. 12–23, Apr 1999.

[43] L. Spainhower and T. Gregg, "IBM S/390 Parallel Enterprise Server G5 Fault Tolerance: A Historical Perspective," *IBM Journal of Research and Development*, vol. 43, no. 5/6, pp. 863–874, 1999.

[44] W. Bartlett and L. Spainhower, "Commercial Fault Tolerance: A Tale of Two Systems," *IEEE Transactions Dependable Secur. Comput.*, vol. 1, no. 1, pp. 87–96, Jan 2004.

[45] N. Saxena, C. Chen, R. Swami, H. Osone, S. Thusoo, D. Lyon, D. Chang, A. Dharmaraj, N. Patkar, Y. Lu, and B. Chia, "Error Detection and Handling in a Superscalar, Speculative Out-of-Order Execution Processor System," *FTCS-25*, pp. 464–471, Jun 1995.

[46] W. W. Hwu and Y. N. Patt, "Checkpoint Repair for Out-Of-Order Execution Machines," in *ISCA-87: Proc. 14th Annual Int. Symp. on Computer architecture*. New York, NY, USA: ACM Press, 1987, pp. 18–26.

[47] S. Kaneda, "A Class of Odd-Weight-Column SEC-DED-SbED Codes for Memory System Applications." *IEEE Transactions on Computers*, vol. 33, no. 8, pp. 737–741, 1984.

[48] N. R. Saxena, C.-W. D. Chang, K. Dawallu, J. Kohli, and P. Helland, "Fault-Tolerant Features in the HaL Memory Management Unit," *IEEE Transactions on Computers*, vol. 44, no. 2, pp. 170–180, Feb 1995.

[49] A. Avizienis, "The Hundred Year Spacecraft," *Proc. First NASA/DoD Workshop on Evolvable Hardware*, pp. 233–239, Jul 1999.

[50] L. Chen and A. Avizienis, "N-version Programming: A Fault Tolerance Approach to Reliability of Software Operation," *FTCS-8*, pp. 3–9, 1978.

[51] C. E. Stroud, "Reliability of Majority Voting Based VLSI Fault-Tolerant Circuits," in *IEEE Transactions on VLSI Systems*, vol. 2, no. 4. IEEE Computer Society, 1994, pp. 516–521.

[52] C. Metra, M. Favalli, and B. Ricco, "Compact and Low Power On-Line Self-Testing Voting Scheme," *DFT-97*, vol. 00, pp. 137–145, Oct 1997.

[53] J. Cazeaux, D. Rossi, and C. Metra, "Self-Checking Voter for High Speed TMR Systems," *Journal of Electronic Testing*, vol. 21, no. 4, pp. 377–389, Aug 2005.

[54] H. El-Razouk and Z. Abid, "A New Transistor-Redundant Voter for Defect-Tolerant Digital Circuits," in *Canadian Conf. on Electrical and Computer Engineering*, 2006, pp. 1078–1081.

[55] S. Mitra and E. J. McCluskey, "Word voter: A new voter design for triple modular redundant systems," *VTS-00: Proc. 18th IEEE VLSI Test Symposium*, pp. 465–470, 2000.

[56] Reynolds and G. Metze, "Fault Detection Capabilities of Alternating Logic," *IEEE Transactions on Computers*, vol. C-27, no. 12, pp. 1093–1098, Dec 1978.

[57] J. Shedletsky, "Error Correction by Alternate-Data Retry," *IEEE Transactions on Computers*, vol. C-27, no. 2, pp. 106–112, Feb 1978.

[58] J. Patel and L. Fung, "Concurrent Error Detection in ALUs by Recomputing with Shifted Operands," *IEEE Transactions on Computers*, vol. C-31, no. 7, pp. 589–595, Jul 1982.

[59] J. Li and E. Swartzlander, "Concurrent Error Detection in ALUs by Recomputing with Rotated Operands," *IEEE Int. Workshop on Defect and Fault Tolerance in VLSI Systems*, pp. 109–116, Nov 1992.

[60] B. Johnson, J. Aylor, and H. Hana, "Efficient use of time and hardware redundancy for concurrent error detection in a 32-bit vlsi adder," *IEEE Journal of Solid-State Circuits*, pp. 208–215, 1988.

[61] Y. Hsu and E. Swartzlander, "Time Redundant Error Correcting Adders and Multipliers," *IEEE Int. Workshop on Defect and Fault Tolerance in VLSI Systems*, pp. 247–256, Nov 1992.

[62] T. Ngai, C. He, and E. Swartzlander, "Enhanced Concurrent Error Correcting Arithmetic Unit Design Using Alternating Logic," in *Proc. 2001 IEEE Int. Symp. on Defect and Fault Tolerance in VLSI Systems*. Washington, DC, USA: IEEE Computer Society, Oct 2001, pp. 78–83.

[63] K-H.Huang and J.A.Abraham, "Algorithm-Based Fault Tolerance for Matrix Operations," *IEEE Transactions on Computers*, vol. C-33, no. 6, pp. 518–528, Jun 1984.

[64] J. Holm and P. Banerjee, "Low Cost Concurrent Error Detection in a VLIW Architecture Using Replicated Instructions," in *ICCP-21: Proc. Int. Conf. on Parallel Processing*, 1992, pp. 192–195.

[65] C. Bolchini, "A Software Methodology for Detecting Hardware Faults in VLIW Data Paths," *IEEE Transactions on Reliability*, vol. 52, no. 4, pp. 458–468, Dec 2003.

[66] J. Hu, F. Li, V. Degalahal, M. Kandemir, N. Vijaykrishnan, and M. Irwin, "Compiler-Directed Instruction Duplication for Soft Error Detection," in *DATE-05: Proc. of the Design, Automation and Test in Europe*, Mar 2005, pp. 1056–1057.

[67] M. Schuette and J. Shen, "Exploiting Instruction-Level Parallelism for Integrated Control-Flow Monitoring," *IEEE Transactions on Computers*, vol. 43, no. 2, pp. 129–140, 1994.

[68] G. Sohi, M. Franklin, and K. Saluja, "A Study of Time-Redundant Fault Tolerance Techniques for High-Performance Pipelined Computers," in *FTCS-19*. Washington, DC, USA: IEEE Computer Society, Jun 1989, pp. 436–443.

[69] M. Franklin, "A Study of Time Redundant Fault Tolerance Techniques for Superscalar Processors," *Proc. IEEE Int. Workshop on Defect and Fault Tolerance in VLSI Systems*, pp. 207–215, Nov 1995.

[70] ——, "Incorporating Fault Tolerance in Superscalar Processors," *HiPC-96: Proc. Int. Conf. on High-Performance Computing*, vol. 00, pp. 301–306, 1996.

[71] T. Austin, "DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design," in *MICRO-32: Proc. 32nd Annual ACM/IEEE Int. Symp. on Microarchitecture*, Washington, DC, USA, Jun 1999, pp. 196–207.

[72] F. Rashid, K. Saluja, and P. Ramanathan, "Fault Tolerance Through Re-execution in Multiscalar Architecture," *Dependable Systems and Networks*, pp. 482–491, 2000.

[73] A. Mendelson and N. Suri, "Designing High-Performance & Reliable Superscalar Architectures: The Out of Order Reliable Superscalar (O3RS) Approach," *Dependable Systems and Networks*, pp. 473–481, Jun 2000.

[74] J. Ray, J. Hoe, and B. Falsafi, "Dual Use of Superscalar Datapath for Transient Fault Detection and Recovery," *MICRO-34*, pp. 214–224, Dec 2001.

[75] J. Smolens, J. Kim, J. Hoe, and B. Falsafi, "Efficient Resource Sharing in Concurrent Error Detecting Superscalar Microarchitectures," *MICRO-37: Proc. of the 37th IEEE/ACM Int. Symp. on Microarchitecture*, vol. 0, pp. 257–268, 2004.

[76] A. Parashar, S. Gurumurthi, and A. Sivasubramaniam, "A Complexity-Effective Approach to ALU Bandwidth Enhancement for Instruction-Level Temporal Redundancy," in *ISCA-04: Proc. of the 31st Annual Int. Symp. on Computer Architecture*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 376–386.

[77] M. Gomaa and T. Vijaykumar, "Opportunistic Transient Fault Detection," *ISCA-05: Proc. 32nd Annual Int. Symp. on Computer Architecture*, pp. 172–183, Jun 2005.

[78] D. Borodin, B. Juurlink, and S. Kaxiras, "Instruction Precomputation for Fault Detection," in *DSD-2009: 12th Euromicro Conf. on Digital System Design*, Aug 2009, pp. 91–99.

[79] S. Kumar and A. Aggarwal, "Self-Checking Instructions – Reducing Instruction Redundancy for Concurrent Error Detection," in *PACT-06: Proc. of the 15th Int. Conf. on Parallel Architectures and Compilation Techniques*. New York, NY, USA: ACM, Sep 2006, pp. 64–73.

[80] D. Borodin, B. Juurlink, and S. Vassiliadis, "Instruction-Level Fault Tolerance Configurability," in *IC-SAMOS VII: Proc. Int. Conf. on Embedded Computer Systems: Architectures, Modeling, and Simulation*, July 2007, pp. 110–117.

[81] E. Rotenberg, "AR-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessors." in *FTCS-29*, Madison, Wisconsin, USA, Jun 1999, pp. 84–91.

[82] S. Reinhardt and S. Mukherjee, "Transient Fault Detection via Simultaneous Multithreading," in *ISCA-00: Proc. 27th Annual Int. Symp. on Computer architecture*, New York, NY, USA, 2000, pp. 25–36.

[83] T. N. Vijaykumar, I. Pomeranz, and K. Cheng, "Transient Fault Recovery Using Simultaneous Multithreading," in *ISCA-02: Proc. 29th Annual Int. Symp. on Computer architecture*, Washington, DC, USA, 2002, pp. 87–98.

[84] S. S. Mukherjee, M. Kontz, and S. K. Reinhardt, "Detailed Design and Evaluation of Redundant Multithreading Alternatives," in *ISCA-02: Proc. 29th Annual Int. Symp. on Computer architecture*. Washington, DC, USA: IEEE Computer Society, 2002, pp. 99–110.

[85] K. Sundaramoorthy, Z. Purser, and E. Rotenberg, "Slipstream Processors: Improving Both Performance and Fault Tolerance," *ACM SIGPLAN Notices*, vol. 35, no. 11, pp. 257–268, 2000.

[86] Z. Purser, K. Sundaramoorthy, and E. Rotenberg, "A Study of Slipstream Processors," in *MICRO-33: Proc. 33rd Annual ACM/IEEE Int. Symp. on Microarchitecture*, New York, NY, USA, 2000, pp. 269–280.

[87] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar, "Multiscalar Processors," in *ISCA-95: Proc. 22nd Annual Int. Symp. on Computer architecture*. New York, NY, USA: ACM Press, Jun 1995, pp. 414–425.

[88] S. Chatterjee, C. Weaver, and T. Austin, "Efficient Checker Processor Design," in *MICRO-33: Proc. 33rd Annual ACM/IEEE Int. Symp. on Microarchitecture*, New York, NY, USA, 2000, pp. 87–97.

[89] M. Mneimneh, F. Aloul, C. Weaver, S. Chatterjee, K. Sakallah, and T. Austin, "Scalable Hybrid Verification of Complex Microprocessors," in *DAC-01: Proc. 38th Conf. on Design automation*. New York, NY, USA: ACM Press, 2001, pp. 41–46.

[90] C. Weaver and T. Austin, "A Fault Tolerant Approach to Microprocessor Design," *Dependable Systems and Networks*, pp. 411–420, Jul 2001.

[91] N. Saxena and E. McCluskey, "Dependable Adaptive Computing Systems–the ROAR project," in *Proc. IEEE Systems, Man, and Cybernetics Conf*, vol. 3, Oct 1998, pp. 2172–2177.

[92] D. Citron, D. Feitelson, and L. Rudolph, "Accelerating Multi-Media Processing by Implementing Memoing in Multiplication and Division Units," *ASPLOS-VIII: Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, vol. 32, no. 5, pp. 252–261, 1998.

[93] D. Borodin and B. Juurlink, "Instruction Precomputation with Memoization for Fault Detection," in *DATE-2010: Proc. of the Design, Automation and Test in Europe*, Mar 2010.

[94] D. Borodin, B. Juurlink, S. Hamdioui, and S. Vassiliadis, "Instruction-Level Fault Tolerance Configurability," *Journal of Signal Processing Systems*, vol. 57, no. 1, pp. 89–105, October 2009.

[95] N. Oh, P. Shirvani, and E. McCluskey, "Error Detection by Duplicated Instructions in Super-Scalar Processors," *IEEE Transactions on Reliability*, vol. 51, no. 1, pp. 63–75, Mar 2002.

[96] D. Tullsen, S. Eggers, and H. Levy, "Simultaneous Multithreading: Maximizing On-Chip Parallelism," in *ISCA-95: Proc. 22nd Annual Int. Symp. on Computer architecture*, New York, NY, USA, 1995, pp. 392–403.

[97] E. Rotenberg, Q. Jacobson, Y. Sazeides, and J. Smith, "Trace Processors," in *MICRO-30: Proc. 30th Annual ACM/IEEE Int. Symp. on Microarchitecture*. Washington, DC, USA: IEEE Computer Society, 1997, pp. 138–148.

[98] K. Olukotun, B. Nayfeh, L. Hammond, K. Wilson, and K. Chang, "The Case for a Single-Chip Multiprocessor," in *ASPLOS-VII: Proc. seventh Int. Conf. on Architectural support for programming languages and operating systems*, New York, NY, USA, 1996, pp. 2–11.

[99] E. Rotenberg, "Exploiting Large Ineffectual Instruction Sequences," Department of Electrical and Computer Engineering, North Carolina State University, Tech. Rep., Nov 1999, available at http://www.tinker.ncsu.edu/ericro/ (last accessed 09.10.2006).

[100] B. Randell, "System Structure for Software Fault Tolerance," in *Proc. Int. Conf. on Reliable software*. New York, NY, USA: ACM Press, 1975, pp. 437–449.

[101] C. C. Li and W. K. Fuchs, "CATCH - Compiler-Assisted Techniques for Checkpointing," in *FTCS-20*. Washington, DC, USA: IEEE Computer Society, 1990, pp. 74–81.

[102] J. Long, W. K. Fuchs, and J. A. Abraham, "Compiler-Assisted Static Checkpoint Insertion," in *FTCS-22*. Washington, DC, USA: IEEE Computer Society, July 1992, pp. 58–65.

[103] M. A. Schuette and J. P. Shen, "Processor control flow monitoring using signatured instruction streams," *IEEE Transactions Computer*, vol. 36, no. 3, pp. 264–277, 1987.

[104] L.McFearin and V.S.S.Nair, "Control-Flow Checking Using Assertions," in *Proc. IFIP Int. Conf. Dependable Computing for Critical Applications*, Sep 1995.

[105] Z. Alkhalifa, V. Nair, N. Krishnamurthy, and J. Abraham, "Design and Evaluation of System-Level Checks for On-Line Control Flow Error Detection," *IEEE Transactions on Parallel and Distributed Systems*, vol. 10, no. 6, pp. 627–641, 1999.

[106] N. Oh, P. Shirvani, and E. McCluskey, "Control-Flow Checking by Software Signatures," *IEEE Transactions on Reliability*, vol. 51, no. 2, pp. 111–122, Mar 2002.

[107] R. Venkatasubramanian, J. Hayes, and B. Murray, "Low-Cost On-Line Fault Detection Using Control Flow Assertions," in *IOLTS: 9th IEEE On-Line Testing Symp.*, Jul 2003, pp. 137–143.

[108] O. Goloubeva, M. Rebaudengo, M. Reorda, and M. Violante, "Soft-Error Detection Using Control Flow Assertions," in *DFT-03: Proc. of the 18th IEEE Int. Symp. on Defect and Fault Tolerance in VLSI Systems*. Washington, DC, USA: IEEE Computer Society, 2003, p. 581.

[109] G. Miremadi, J.Harlsson, U. Gunneflo, and J.Torin, "Two Software Techniques for On-Line Error Detection," in *FTCS-22*. Washington, DC, USA: IEEE Computer Society, 1992, pp. 328–335.

[110] J. M. Ayache, P. Azema, and M. Diaz, "Observer: a Concept for On-Line Detection of Control Errors in Concurrent Systems," in *FTCS-9*. Washington, DC, USA: IEEE Computer Society, 1979, pp. 79–86.

[111] S. S. Yau and F. C. Chen, "An Approach to Concurrent Control Flow Checking," *IEEE Transactions on Software Engineering*, vol. SE-6, no. 2, pp. 126–137, 1980.

[112] D. Lu, "Watchdog Processors and Structural Integrity Checking," *IEEE Transactions on Computers*, vol. C-31, no. 7, pp. 681–685, Jul 1982.

[113] T. Michel, R. Leveugle, and G. Saucier, "A New Approach to Control Flow Checking Without Program Modification," in *FTCS-21: The Twenty-First Int. Symp. on Fault-Tolerant Computing*, Jun 1991, pp. 334–341.

[114] M. Rebaudengo, M. S. Reorda, M. Torchiano, and M. Violante, "Soft-Error Detection through Software Fault-Tolerance Techniques," *DFT-99*, pp. 210–218, 1999.

[115] ——, "A Source-to-Source Compiler for Generating Dependable Software," *First IEEE Int. Workshop on Source Code Analysis and Manipulation*, pp. 33–42, 2001.

[116] B. Nicolescu and R. Velazco, "Detecting Soft Errors by a Purely Software Approach: Method, Tools and Experimental Results," in *DATE-03: Proc. of the Design, Automation and Test in Europe Conference and Exhibition*, 2003, pp. 57–62.

[117] N. Oh and E. McCluskey, "Error Detection by Selective Procedure Call Duplication for Low Energy Consumption," *IEEE Transactions on Reliability*, vol. 51, no. 4, pp. 392–402, Dec 2002.

[118] G. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. August, "SWIFT: Software Implemented Fault Tolerance," in *CGO-05: Proc. of the Int. Symp. on Code Generation and Optimization*, Washington, DC, USA, 2005, pp. 243–254.

[119] G. Reis, J. Chang, N. Vachharajani, R. Rangan, D. August, and S. Mukherjee, "Design and Evaluation of Hybrid Fault-Detection Systems," *ISCA-05: Proc. 32nd Annual Int. Symp. on Computer Architecture*, vol. 0, pp. 148–159, Jun 2005.

[120] N. Oh, S. Mitra, and E. McCluskey, "ED^4I: Error Detection by Diverse Data and Duplicated Instructions," *IEEE Transactions on Computers*, vol. 51, no. 2, pp. 180–199, 2002.

[121] P. Shirvani, N. R. Saxena, and E. J. McCluskey, "Software-Implemented EDAC Protection Against SEUs," *IEEE Transactions on Reliability*, vol. 49, no. 3, pp. 273–284, Sep 2000.

[122] G. K. Saha, "Software Based Fault Tolerant Computing," *ACM Ubiquity*, vol. 6, no. 40, pp. 1–1, 2005.

[123] M. Namjoo and E. J. McCluskey, "Watchdog Processors and Capability Checking," in *FTCS-12*. Washington, DC, USA: IEEE Computer Society, 1982, pp. 245–248.

[124] Y. Huang and C. Kintala, "Software Implemented Fault Tolerance: Technologies and Experience," in *FTCS-23*. Washington, DC, USA: IEEE Computer Society, 1993, pp. 2–9.

[125] J. Hennessy and D. Patterson, *Computer Architecture, a Quantitative Approach*, 3rd ed. Morgan Kaufmann, May 2003.

[126] A. Mahmood and E. McCluskey, "Concurrent Error Detection Using Watchdog Processors–A Survey," *IEEE Transactions on Computers*, vol. 37, no. 2, pp. 160–174, Feb 1988.

[127] A. Avizienis, "Fault Tolerance by Design Diversity: Concepts and Experiments," *IEEE Computer*, vol. 17, no. 8, pp. 67–80, Aug 1984.

[128] S. Borkar, P. Dubey, K. Kahn, D. Kuck, H. Mulder, S. Pawlowski, and J. Rattner, "Platform 2015: Intel Processor and Platform Evolution for the Next Decade," *Technology@Intel Magazine*, Mar 2005, available at http://http://www.intel.com/technology/magazine/computing/platform-2015-0305.htm/.

[129] J. F. Cantin, M. H. Lipasti, and J. E. Smith, "Dynamic Verification of Cache Coherence Protocols," in *Proc. ISCA Workshop on Memory Performance Issues*. New York, NY, USA: ACM Press, Jun 2001.

[130] A. Meixner and D. Sorin, "Error Detection via Online Checking of Cache Coherence with Token Coherence Signatures," in *HPCA-2007: Proc. IEEE 13th Int. Symp. on High Performance Computer Architecture*, Feb 2007, pp. 145–156.

[131] M. Martin, M. Hill, and D. Wood, "Token Coherence: Decoupling Performance and Correctness," in *ISCA: Proc. 30th Annual Int. Symp. on Computer Architecture*, Jun 2003, pp. 182–193.

[132] D. Sorin, M. Hill, and D. Wood, "Dynamic Verification of End-to-End Multiprocessor Invariants," in *DSN-03: Proc. Int. Conf. on Dependable Systems and Networks*, Jun 2003, pp. 281–290.

[133] D. Borodin and B. Juurlink, "A Low-Cost Cache Coherence Verification Method for Snooping Systems," in *DSD-2008: 11th Euromicro Conf. on Digital System Design*, Sep 2008, pp. 219–227.

[134] R. Fernandez-Pascual, J. Garcia, M. Acacio, and J. Duato, "A Low Overhead Fault Tolerant Coherence Protocol for CMP Architectures," in *HPCA-2007: Proc. IEEE 13th Int. Symp. on High Performance Computer Architecture*, Feb 2007, pp. 157–168.

[135] M. Breuer, S. Gupta, and T. Mak, "Defect and Error Tolerance in the Presence of Massive Numbers of Defects," *IEEE Design and Test of Computers*, vol. 21, no. 3, pp. 216–227, 2004.

[136] H. Chung and A. Ortega, "Analysis and Testing for Error Tolerant Motion Estimation," in *DFT-05: Proc. 20th IEEE Int. Symp. on Defect and Fault Tolerance in VLSI Systems*, Washington, DC, USA, Oct 2005, pp. 514–522.

[137] F. K. Jondral, "Software-Defined Radio: Basics and Evolution to Cognitive Radio," *EURASIP Journal on Wireless Communications and Networking*, vol. 2005, no. 3, pp. 275–283, 2005.

[138] "Fibonacci numbers at Wikipedia," http://en.wikipedia.org/wiki/Fibonacci_number.

[139] D. Burger and T. Austin, "The SimpleScalar Tool Set, Version 2.0," *SIGARCH Comput. Archit. News*, vol. 25, no. 3, pp. 13–25, 1997.

[140] T. Austin, E. Larson, and D. Ernst, "SimpleScalar: An Infrastructure for Computer System Modeling," *IEEE Computer*, vol. 35, no. 2, pp. 59–67, 2002.

[141] D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: A Framework for Architectural-Level Power Analysis and Optimizations," in *ISCA-00: Proc. of the 27th Annual Int. Symp. on Computer Architecture*, New York, NY, USA, 2000, pp. 83–94. [Online]. Available: citeseer.ist.psu.edu/brooks00wattch.html

[142] "Independent JPEG Group webpage," http://www.ijg.org/.

[143] A. Sundaram, A. Aakel, D. Lockhart, D. Thaker, and D. Franklin, "Efficient Fault Tolerance in Multi-Media Applications through Selective Instruction Replication," in *WREFT-08: Proc. of the 2008 workshop on Radiation effects and fault tolerance in nanometer technologies*. New York, NY, USA: ACM, 2008, pp. 339–346.

[144] S. Mukherjee, C. Weaver, J. Emer, S. Reinhardt, and T. Austin, "A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-Performance Microprocessor," in *MICRO-36: Proc. of the 36th Annual IEEE/ACM Int. Symp. on Microarchitecture*. Washington, DC, USA: IEEE Computer Society, 2003, p. 29.

[145] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communicatons Systems," in *MICRO-30: Proc. of the 30th Annual ACM/IEEE Int. Symp. on Microarchitecture*. Washington, DC, USA: IEEE Computer Society, 1997, pp. 330–335.

[146] P. Norvig, "Techniques for Automatic Memoization with Applications to Context-Free Parsing," *Computational Linguistics*, vol. 17, no. 1, pp. 91–98, Mar 1991.

[147] S. F. Oberman and M. J. Flynn, "Reducing Division Latency with Reciprocal Caches," *Reliable Computing*, vol. 2, no. 2, pp. 147–153, Apr 1996.

[148] J. L. Henning, "SPEC CPU2000: Measuring CPU Performance in the New Millennium," *IEEE Computer*, vol. 33, no. 7, pp. 28–35, 2000.

[149] "SPEC CPU2000 benchmark suite," http://www.spec.org/cpu2000/.

[150] M. Papamarcos and J. Patel, "A Low-Overhead Coherence Solution for Multiprocessors with Private Cache Memories," in *ISCA-84: Proc. of the 11th Annual Int. Symp. on Computer Architecture*. New York, NY, USA: ACM Press, 1984, pp. 348–354.

[151] D. Sorin, M. Martin, M. Hill, and D. Wood, "SafetyNet: Improving the Availability of Shared Memory Multiprocessors with Global Checkpoint/Recovery," in *ISCA-02: Proc. 29th Annual Int. Symp. on Computer Architecture*, 2002, pp. 123–134.

[152] M. Prvulovic, Z. Zhang, and J. Torrellas, "ReVive: Cost-Effective Architectural Support for Rollback Recovery in Shared-Memory Multiprocessors," in *ISCA-02: Proc. 29th Annual Int. Symp. on Computer Architecture*, 2002, pp. 111–122.

[153] D. August, J. Chang, S. Girbal, D. Gracia-Perez, G. Mouchard, D. A. Penry, O. Temam, and N. Vachharajani, "UNISIM: An Open Simulation Environment and Library for Complex Architecture Design and Collaborative Development," *IEEE Computer Architecture Letters*, vol. 6, no. 2, pp. 45–48, 2007.

[154] "UNISIM: UNIted SIMulation environment webpage," http://unisim.org/.

List of Publications

International Journals

1. Demid Borodin, B.H.H. (Ben) Juurlink, Said Hamdioui, and Stamatis Vassiliadis, **Instruction-Level Fault Tolerance Configurability**, *Journal of Signal Processing Systems*, Volume 57, Issue 1, pp. 89–105, October 2009.

2. Asadollah Shahbahrami, B.H.H. (Ben) Juurlink, Demid Borodin, and Stamatis Vassiliadis, **Avoiding Conversion and Rearrangement Overhead in SIMD Architectures**, *International Journal of Parallel Programming*, Volume 34, Issue 3, pp. 237-260, June 2006.

International Conferences

1. Demid Borodin and B.H.H. (Ben) Juurlink, **Protective Redundancy Overhead Reduction Using Instruction Vulnerability Factor**, *Proceedings of the ACM International Conference on Computing Frontiers*, May 2010.

2. Demid Borodin and B.H.H. (Ben) Juurlink, **Instruction Precomputation with Memoization for Fault Detection**, *DATE'2010: Proceedings of the Design, Automation and Test in Europe*, March 2010.

3. Demid Borodin, B.H.H. (Ben) Juurlink, and Stefanos Kaxiras, **Instruction Precomputation for Fault Detection**, *DSD'2009: Proceedings of the 12th Euromicro Conference on Digital System Design*, pp. 91–99, August 2009.

4. Demid Borodin and B.H.H. (Ben) Juurlink, **A Low-Cost Cache Coherence Verification Method for Snooping Systems**, *DSD'2008: Proceedings of the 11th Euromicro Conference on Digital System Design*, pp. 219–227, September 2008.

5. Demid Borodin, B.H.H. (Ben) Juurlink, and Stamatis Vassiliadis, **Instruction-Level Fault Tolerance Configurability**, *IC-SAMOS VII: International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation*, pp. 110–117, July 2007.

Local Conferences

1. Asadollah Shahbahrami, Demid Borodin, and B.H.H. (Ben) Juurlink, **Comparison Between Color and Texture Features for Image Retrieval**, *ProRisc'2008: Proceedings of the 19th Annual Workshop on Circuits, Systems and Signal Processing*, November 2008.

2. Carsten M. van der Hoeven, B.H.H. (Ben) Juurlink, and Demid Borodin, **SimpleScalar Macro Tool**, *ProRISC'2007: Proceedings of the 18th Annual Workshop on Circuits, Systems and Signal Processing*, November 2007.

3. Demid Borodin, Andrei Terechko, B.H.H. (Ben) Juurlink, and Paulus Stravers, **Optimisation of Multimedia Applications for the Philips Wasabi Multiprocessor System**, *ProRISC'2005: Proceedings of the 16th Annual Workshop on Circuits, Systems and Signal Processing*, November 2005.

Samenvatting

In dit proefschrift richten wij ons op de overheadreductie van fault tolerance (FT) technieken. Door technologische ontwikkelingen, zoals de afnemende afmeting van chip-transistoren en lagere spanningsniveaus, wordt FT steeds belangrijker in moderne computersystemen. FT technieken zijn gebaseerd op bepaalde redundantie vormen. Voorbeelden hiervan zijn: ruimte-redundantie (extra hardware), tijd-redundantie (meerdere uitvoeringen) en/of informatie redundantie (extra verificatie informatie). Door de redundantie worden de kosten hoger en/of leidt dit tot het afnemen van de prestaties en dat is in de meeste gevallen niet aanvaardbaar.

In dit proefschrift worden verschillende methodes voorgesteld die de overhead van de FT technieken reduceren. De meeste technieken hebben tot doel de overhead, veroorzaakt door tijd-redundantie, te reduceren, hoewel sommige FT technieken ook geschikt zijn om de overhead van andere vormen van redundantie te verminderen. Veel tijd-redundantie FT technieken zijn gebaseerd op het herhaaldelijk uitvoeren van instructies. Kopieen van redundante instructies worden in hardware of software gemaakt en hun resultaten worden vergeleken om mogelijke fouten te detecteren.

Dit proefschrift stelt dat er voor verschillende instructies verschillende beschermingsniveaus noodzakelijk zijn voor een betrouwbare uitvoering van toepassingen. Er zijn mogelijke manieren onderzocht die geschikte beschermingsniveaus toepassen op de verschillende instructies, bijvoorbeeld met behulp van het voorgestelde nieuwe concept van de instructiekwetsbaarheidsfactor. Door kritieke instructies beter te berschermen kan men een aanzienlijke prestatieverbetering, energieverbruik vermindering en/of systeem kostenbesparing bereiken. Bovendien worden “instruction reuse” techieken, precomputation en memoization aangedragen om het aantal noodzakelijke instructies, die meerdere malen herhaald moeten worden voor de foutdetectie, te reduceren.

Door de recente overtuiging dat instruction level parallelism (ILP) zijn grenzen heeft bereikt en door de beperking van het te gebruiken vermogen, hebben multiprocessorsystemen onlangs veel aandacht gekregen. In een multiprocessorsysteem met cache coherency, is het correct functioneren van het cache coherency protocol van essentieel belang voor het systeem. Dit proefschrift presenteert een cache coherence verificatie techniek die in vergelijking met eerder voorgestelde methodes tegen lagere kosten fouten detecteert.

Curriculum Vitae

Demid Borodin was born in Samarkand, USSR, in 1982. In 2003 he received the BSc degree in aerospace engineering from Tashkent State Institute of Aviation, Tashkent, Uzbekistan. In 2005 he received the MSc degree in computer engineering from Delft University of Technology, Delft, The Netherlands.

Subsequently in 2005 he continued to work in the Computer Engineering Laboratory as a PhD student, under the guidance of Dr. Ben Juurlink. Currently he is pursuing postdoctoral research at the same laboratory. The PhD research focuses on fault tolerance of computing systems. The interests also include application-specific instruction set architectures, parallel architectures, three-dimensional integrated circuits, and computer architecture in general.