UNIVERSIDAD DE MURCIA

Facultad de Informática

Departamento de Ingeniería y Tecnología de Computadores

Microarchitectural Approaches for Hardware Fault Mitigation in Multicore Processors

A dissertation submitted in fulfillment of
the requirements for the degree of

DOCTOR EN INFORMÁTICA

by

Daniel Sánchez

Advised by:

Juan Luis Aragón

José Manuel García

Murcia, June, 2011

UNIVERSIDAD
DE MURCIA

DEPARTAMENTO DE INGENIERÍA Y TECNOLOGÍA DE COMPUTADORES

D. Juan Luis Aragón Alcaraz, Profesor Titular de Universidad del Área de Arquitectura y Tecnología de Computadores en el Departamento de Ingeniería y Tecnología de Computadores de la Universidad de Murcia

y

D. José Manuel García Carrasco, Catedrático de Universidad del Área de Arquitectura y Tecnología de Computadores en el Departamento de Ingeniería y Tecnología de Computadores de la Universidad de Murcia

AUTORIZAN:

La presentación de la Tesis Doctoral titulada «*Diseño de Arquitecturas para la Mitigación de Fallos Hardware en Procesadores Multinúcleo*», realizada por D. Daniel Sánchez Pedreño, bajo su inmediata dirección y supervisión, y que presenta para la obtención del grado de Doctor Europeo por la Universidad de Murcia.

En Murcia, a 8 de Abril de 2011.

Fdo: Dr. Juan Luis Aragón Alcaraz

Fdo: Dr. José Manuel García Carrasco

UNIVERSIDAD DE
MURCIA

D. Antonio Fernando Gómez Skarmeta, Catedrático de Universidad del Área de Ingeniería Telemática y presidente de la Comisión Académica del Programa de Postgrado de Tecnologías de la Información y Telemática Avanzadas de la Universidad de Murcia, INFORMA:

Que la Tesis Doctoral titulada «*Diseño de Arquitecturas para la Mitigación de Fallos Hardware en Procesadores Multinúcleo*», ha sido realizada por D. Daniel Sánchez Pedreño, bajo la inmediata dirección y supervisión de D. Juan Luis Aragón Alcaraz y de D. José Manuel García Carrasco, y que la Comisión Académica ha dado su conformidad para que sea presentada ante la Comisión de Doctorado.

En Murcia, a 8 de Abril de 2011.

Fdo: Dr. Antonio Fernando Gómez Skarmeta

Dedicado a H

Abstract

So far, improvements in the CMOS scaling fabrication technology has permitted to exponentially increase the number of transistors per chip. For some time, this increasing count of transistors was employed for the design of more aggressive out-of-order single-core processors. However, at the time that proposed techniques were not able to exploit the Instruction Level Parallelism (ILP), architectural designs were forced to move towards multi-core architectures to exploit Thread Level Parallelism (TLP), as a way of delivering an increased performance while maintaining power and design complexity at manageable levels. Nonetheless, as a counterpart, miniaturization trends are increasing the susceptibility of integrated circuits to a variety of hardware errors such as soft errors, wear-out related permanent faults and process variations.

In this Thesis, we have focused on architectural level mechanisms to deal with this increasing fault rate and variation. First, we deal with the arise of transient faults in shared-memory architectural designs. To that end, based on Redundant Multi Threading previous approaches, we propose *Reliable Execution of Parallel ApplicationS in tiled-CMPs* (REPAS). In this approach, redundant threads are executed in the same SMT core as a way to reduce the hardware overhead while incurring into a moderate performance slowdown. In REPAS, we address the under-explored support for the correct execution of shared-memory parallel applications.

Afterwards, as a way to minimize the increased complexity, hardware and performance overhead, we present *Log-Based Redundant Architecture* (LBRA), a highly decoupled redundant architecture based on a hardware transactional memory implementation. We leverage the already introduced hardware of LogTM-SE to provide a consistent view of the memory between master and slave threads through a virtualized memory log, both transient fault detection and recovery, more scalability, higher decoupling and lower performance overhead than previous proposals.

Finally, we study the impact of hard faults on cache memories. To this end, we develop an analytical model for determining the implications of word/block disabling techniques due to random cell failure on cache miss rate behaviour. The proposed model is distinct from previous work in that it is an exact model rather than an approximation. Besides, it is simpler than previous experimental frameworks which are based on the use of fault maps as a brute force approach to statistically approximate the effect of random cell failure on caches.

Agradecimientos

Nadie dijo que fuera a ser fácil.
Nadie dijo que fuese a ser así de difícil.

Después de mucho trabajo a lo largo de estos años, al fin puedo decir bien alto: ¡lo hice!. Así que, tras la escitura de esta Tesis, y comoquiera que ya he plantado un árbol, parece que ya solo queda por solventar el tema del hijo... . En fin, todo se andará.

En primer lugar me gustaría agradecer la confianza que hace ya algunos años mis directores José Manuel y Juan Luis depositaron en mí. A lo largo de estos años han sido a la vez una guía y una fuente de estímulo. También quería agradecerles su comprensión cuando las cosas no salían a la primera y por el aliento que recibí de ellos para intentar hacer las cosas cada vez mejor.

En cualquier caso, todo esto nunca hubiera sido posible sin el inestimable apoyo, en todos los sentidos, que me han dado mis padres. Sin la educación que he recibido de vosotros jamás habría llegado hasta aquí. Ya se que nunca he sido el hijo más comunicativo del mundo (más bien todo lo contrario), por eso quiero que sirvan estas palabras para agradeceros todo lo que habéis hecho por mi y deciros que siempre vais a ser un referente. De la misma forma querría acordarme de mi hermana, que ha tenido que lidiar conmigo desde que nació. También un recuerdo especial para toda mi familia: tíos, tías, primos, primas, y especialmente a mi abuela, que ya tiene un nieto doctor (aunque sea de los de "mentira").

Apartado especial merece mi novia Helena, a la que dedico esta Tesis. Mi familia no tiene más remedio que aguantarme, claro, pero lo tuyo es de traca. Siempre has estado ahí, incluso a pesar de los kilómetros. Sabiendo cómo estaba con tan solo pronunciar una palabra. Haciéndome feliz, vaya. Pero, además, tu contribución a esta Tesis ha ido un paso más allá, ya que por tus manos han pasado todos mis artículos para pulirme esos fallos tan grotescos que cometo con el inglés. Así que gracias dobles, por lo personal y lo profesional.

También me gustaría acordarme de mis mejores amigos: Jesula, Chavo, Laura, Guillermo, Jon y el resto del grupo (vosotros sabéis quienes sois), que desde el instituto e incluso antes forman una parte muy importante de mi vida. Y por supuesto, a todos mis amigos y compañeros del laboratorio que, en realidad, son como una segunda familia: Juan Manuel, Rubén, Kenneth, Chema, Antonio y Ginés. Y también a Ricardo y Alberto, que nos han ido abriendo camino y haciendo que las cosas sean más fáciles para los que hemos ido detrás.

De la misma manera, querría recordar a toda la gente que conocí durante mi estancia en Chipre. *Thank you Yanos for your valuable support. It was just three months but I learned a lot. I would also like to give my best regards to my fellows Isidoros, Bushra, Nikolas, Lorena and all the people I met in the UCY. Thank you all for treating me so well.*

No me olvidaré tampoco de toda la gente que ha contribuido a mi formación, desde mis "maestros" del C.P. "La Concepción" y los distintos profesores del Instituto "Jiménez de la Espada" de Cartagena así como de mis profesores y compañeros de la Facultad de Informática de la Universidad de Murcia. Vuestro esfuerzo y dedicación, parece, ha dado recompensa.

¡Qué no se me olviden tampoco los agradecimientos oficiales!. Esta Tesis ha sido financiada por el Ministerio de Educación y Ciencia de España y por fondos de la Comisión Europea FEDER en los programas "Consolider Ingenio-2010 CSD2006-00046" y "TIN2009-14475-C04-02". Daniel Sánchez ha sido financiado mediante una beca de investigación del Ministerio de Educación y Ciencia de España en el programa "Consolider Ingenio-2010 CSD2006-00046". Además, parte de esta Tesis fue elaborada durante una estancia pre-doctoral HiPEAC (FP7 Network of Excellence) de Daniel Sánchez en la Universidad de Chipre.

Y por último gracias a los *anonymous reviewers* por sus comentarios y sugerencias durante todos estos años. Quienquiera que sean.

Contents

List of Figures

List of Tables

Abbreviations

BER	**B**ackward **E**rror **R**ecovery
CARER	**C**ache **A**ided **R**ollback **E**rror **R**ecovery
CMP	**C**hip **M**ulti **P**rocessor
CRC	**C**yclic **R**edundancy **C**ode
DAG	**D**irected **A**cyclic **G**raph
DMR	**D**ual **M**odular **R**edundancy
DRAM	**D**ynamic **R**andom **A**ccess **M**emory
ECC	**E**rror **C**orrecting **C**ode
EMR	**E**xpected **M**iss **R**atio
FIFO	**F**irst **I**n **F**irst **O**ut
HTM	**H**ardware **T**ransactional **M**emory
ILP	**I**nstruction **L**evel **P**arallelism
IPC	**I**nstructions **P**er **C**ycle
LRU	**L**ast **R**ecently **U**sed
MTTF	**M**ean **T**ime **T**o **F**ailure
RMT	**R**edundant **M**ulti **T**hreading
SDC	**S**ilent **D**ata **C**orruption
SD_MR	**S**tandard **D**eviation for **M**iss **R**atio
SECDED	**S**ingle **E**rror **C**orrection / **D**ouble **E**rror **D**etection

SER	**S**oft **E**rror **R**ate
SEU	**S**ingle **E**vent **U**pdate
SOI	**S**ilicon **O**n **I**nsulator
SMN	**S**tatic **M**agnetic **N**oise
SMT	**S**imultaneous **M**ulti **T**hread
SRAM	**S**tatic **R**andom **A**ccess **M**emory
TLP	**T**hread **L**evel **P**arallelism
TMR	**T**riple **M**odular **R**edundancy
TSO	**T**otal **S**tore **O**rder

"We have forty million reasons for failure, but not a single excuse.."

Rudyard Kipling

Chapter 1

Introduction

Nowadays, hardware reliability is considered a first-class issue in processor architecture along with performance and energy efficiency. The increasing scaling technology and subsequent supply voltage reductions, together with temperature fluctuations, augment the susceptibility of architectures to errors.

As systems grow in size and capacity, the transistors employed in their construction become surprisingly smaller, something which makes them more prone to errors. In the past 50 years, improvements in transistor shrinking have allowed to double the capacity and speed of computers every 18 months, according to Moore's Law [52]. However, this has come to a cost: today and tomorrow's transistors have a greater erratic behaviour than their predecessors had. This issue limits, in general terms, the performance of all kinds of computers, from those running applications which require high availability and reliability to even commodity systems. Thus, fault tolerance has a paramount importance in the design and implementation of microarchitectures in the short, medium and long term.

There is a vast amount of work dealing with reliability issues and proposals which, in fact, have been implemented in very specific environments in which the incurred cost, which is either economic, performance slowdown, power increase, or all of them, may be fully justified. However, the arising of hardware faults is no longer just an issue for high specialized domains but for general-purpose

computing systems as well. Thus, we need to find cheaper reliability solutions to avoid these faults or, at least, mitigate their impact.

1.1 Hardware Faults

Hardware structures are subject to *faults* due to defects, imperfections or interactions with the external environment. When a fault has user-visible effects, we call it an *error*. Luckily, not all faults manifest themselves like errors (meaning that outputs of affected devices continue being correct). However, other aspects such as the performance or power dissipation can be affected. We call these hardware faults *benign faults* or *masked faults*. One example of these faults are those regarding prediction units, which have an impact on the performance by miss-predicting a branch, although not affecting the output of the program.

Faults effects can be classified in different ways. One classification distinguishes between logical and parametric faults [76].

- **Logical Faults**. These faults produce errors in the boolean state of devices. Among them, we find *stuck-at faults*, in which devices always provide the same logical value (0 or 1); *von Neumann faults*, in which the output is inverted from the correct value, and *bridging faults*, in which the output of one or more adjacent devices is changed with the value of others due to signal crosstalking.
- **Parametric Faults**. These faults are related to timing issues, which may not induce errors like benign faults. Other faults in this category are current faults, in which the leakage of devices grows abnormally.

Traditionally, hardware errors have been divided into three main categories according to their nature and duration: *transient faults*, *intermittent faults* and *permanent faults*.

1.1.1 Transient Faults

Transient faults are radiation-induced faults which can manifest themselves as transient errors (also known as *soft errors* or SEU). Due to their nature, the occurrence of this kind of faults can be considered as random. Transient faults can be induced by a variety of sources such as transistor variability, thermal cycling, erratic fluctuations of voltage and radiation external to the chip [54]. Radiation-induced events include alpha-particles from packaging materials and neutrons from the atmosphere.

It is well established that the charge of an alpha particle or a neutron strike over a logical device can overwhelm the circuit inducing its malfunction [6, 49]. Strikes create a cylindrical track of electron-hole pairs with very high carrier concentration as is depicted in Figure 1.1. As a result of this impact, extra electrons are collected in the depletion region. When this charge exceeds the Q_{crit} (critical charge), the value stored within the affected memory cell or a transistor can flip [88], inducing a transient fault. After this event, the affected device may continue working properly.

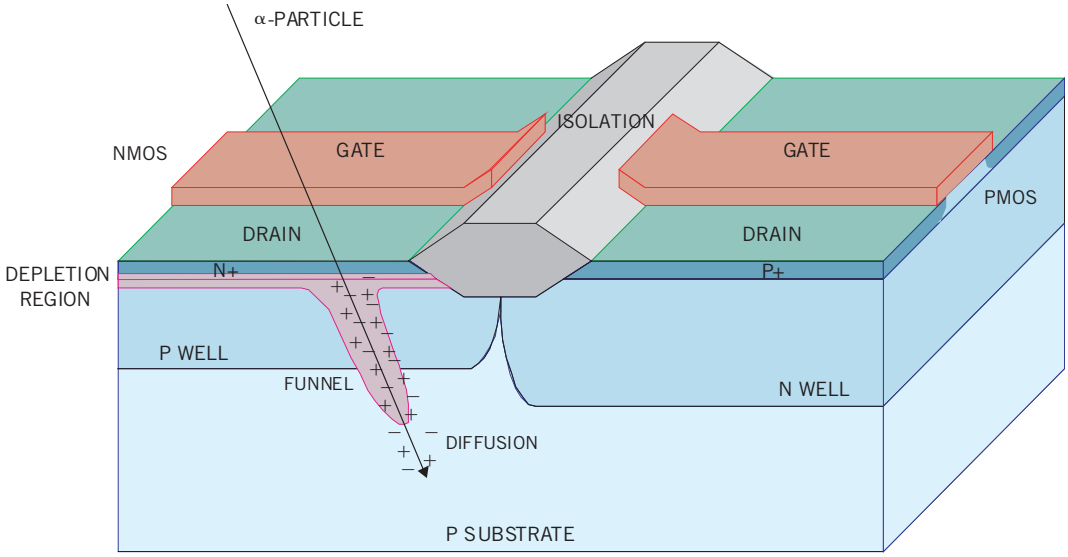


FIGURE 1.1: Radiation particle strike [49].

At the same time, it is also known that the intensity of cosmic rays (high energy radiation striking the Earth from outer space) increases with altitude and

varies with latitude [118]. This, together with their inherent critically, has made that equipment on board of planes and satellites had been traditionally protected to deal with this kind of faults. For instance, the chances of suffering a soft error increases 100 times at 10 km of altitude with respect to the sea level [49, 118].

The occurrence of transient faults in current technologies is the most predominant source for errors in microarchitectures [90]. The use of smaller devices and the decrease of voltage for power dissipation reasons reduces the Q_{crit}, which make them even more prone to transient faults [6, 46, 82].

1.1.2 Intermittent Faults

Intermittent faults are faults which appear and disappear repeatedly but not continuously in time. As well as transient faults, these faults are non-permanent. Errors induced by transient and intermittent faults manifest similarly, although there exists some differences. On the one hand, errors induced by intermittent faults usually occur in bursts when the fault is activated. On the other hand, replacement of the affected device eliminates an intermittent fault, whereas transient faults cannot be fixed or avoided by repair. We find the origin of intermittent faults in voltage peaks, falls and/or temperature fluctuations. As shown recently, intermittent faults are an indicative of prefailure component wear-out [17].

1.1.3 Permanent Faults

Finally, permanent faults, also called *hard faults*, induce permanent errors due to irreversible physical changes. These faults are either occasioned during fabrication process and/or chip operation. Until disabled or repaired, a permanently damaged component will continue producing erroneous results.

Sources for permanent faults can be classified into three main categories [88]:

- **Physical wear-out**. There are several factors which can emphasize aging-related problems. Electromigration [16, 33] is caused by a current flowing

in a conductor moving metal ions until they form *voids*. These voids can either derive into opens or shorts. The *oxide gate breakdown* is another relevant problem [91]. This is defined as the time when there is a conduction path from the anode to the cathode through the gate oxide as a result of the reduced dimensions of gates, which result thinner in every new scale generation in order to facilitate power reduction. Finally, there exists other physical events which can reduce the reliability of devices such as mechanical stress or temperature variations.

- **Fabrication defects**. Fabrication is not a perfect process. Indeed, the number of fabrication defects grows exponentially with every scale generation due to the reduced dimensions of transistors. Defects at manufacturing time causes the same problems as wear-out faults but from the very first moment.

- **Design bugs**. As well as the fabrication process, chips are not perfect in design. Although they are subject to a large amount of tests before arriving to market, sometimes, bugs which were committed during design are not detected. Some bugs which do not affect the behaviour of processors dramatically are sometimes simply ignored or patched by firmware/software solutions. But, in the case of serious bugs, e.g. those related to the correction of computation, vendors have no solution but to retire the shipped products.

1.2 Motivation

The increasing device density offers designers the opportunity to place more functionality per unit area. Billions of transistors are available within a single chip but existing techniques are not available to exploit even more the Instruction Level Parallelism (ILP). The solution designers have adopted is to use this vast amount of transistors to the integration of large caches and many cores into the same chip or Chip Multi Processors (CMP) [62, 77] to exploit the Thread Level Parallelism (TLP). These CMPs are usually implemented around a shared-memory environment, in which some levels of the memory hierarchy are private (typically the L1

cache) although coherent and the rest of the levels are shared. First implementations connect the cores by means of shared-buses and/or crossbars. However, as the number of cores grows, these networks result in non-scalable due to area and power constraints [34]. The most promising approach to provide efficiency and scalability are directory-based cache coherence protocols [14, 18] which operate in direct network environments.

Unfortunately, device area scaling has been accompanied by, at least, two negative consequences: a slowdown in both voltage scaling and frequency increase due to slower scaling of leakage current as compared to area scaling [9, 22, 101], and a shift to probabilistic design and less reliable silicon primitives due to static [10] and dynamic [11] variations.

These alarming trends are leading to forecast that the performance and cost benefits from area scaling will be hindered unless scalable techniques are developed to address power and reliability challenges. In particular, it will be impossible to operate all on-chip resources, even at the minimum voltage for safe operation, due to power constraints, and/or the growing design and operational margins used to provide silicon primitives with resiliency against variations which will consume the scaling benefits.

A modern computer is formed by the processor core logic, main memory (usually DRAM), embedded memory (registers and caches, built with SRAM), an interconnection network and several I/O components. In spite of the random nature of hardware faults, components like the core logic and memories are more affected by faults because they occupy a considerable portion of the total chip area.

The core logic in high-performance systems has traditionally been protected from transient faults by means of TMR approaches. However, this solution is not feasible for wider markets. Industry is aware of the growing trend of transient faults but, so far, few new designs seriously address this problem, since appropriate mechanisms would imply performance degradation, additional circuitry to detect faults and increased manufacturing costs. Therefore, this performance/reliability challenge is addressed by means of new architectural-level designs based

on redundancy. At this level, more flexibility can be provided in comparison to circuit-level, in which measures introduce fixed hardware overheads and limit the implementation in real designs.

On the other hand, embedded memory uses inexpensive ECC codes to detect and correct soft-errors. However, ECC is not a performance friendly mechanism for permanent errors because every access to a faulty block incurs the ECC repair overhead. Furthermore, ECC soft-error capabilities are reduced when some bits protected by the ECC code are already faulty. Thus, ECC may not be the best option to repair permanent or wear-out faults in the cache. It seems then that a better solution for hard fault mitigation in memory structures may be disabling techniques, which try to obtain a benefit from the capacity/performance trade-off.

1.3 Contributions of this Thesis

In this Thesis, we focus on the impact of faults in the core logic and embedded memory. Specifically, we study a) transient fault detection and correction in the core logic, and b) the effect of permanent faults in SRAM arrays. In both cases, the main goal is to reduce both the performance impact and area overhead of different mechanisms in current architectures. In order to mitigate the impact of transient faults, we propose two hardware mechanisms in which computation is checked by means of Redundant Multi Threading (RMT) solutions, a special case of DMR in which fault tolerant computation is accomplished by redundant threads. For dealing with permanent faults in caches, we study the implications of cell/block disabling mechanisms, techniques which rely on the disabling of faulty portions whenever a permanent error is detected.

In summary, the main contributions of this Thesis are:

- We have identified the major drawbacks of previous RMT approaches. In particular, we point out how the support to reliably execute shared-memory applications is not well suited for some previous approaches. In order to avoid inconsistencies in memory accesses between redundant computations,

additional measures must be taken. Unfortunately, these measures have an impact on the overall performance of the architecture.

- A novel hardware RMT mechanism called REPAS which, based on a 2-way Simultaneous Multi Threading (SMT) core architecture, is able to detect and correct transient faults affecting the program output. Previous approaches stall the memory updates until values to store are successfully verified. This imposes several problems regarding the correct execution of shared-memory workloads. Thus, we propose to update memory without verification under the assumption that memory updates are correct (common case). If a fault is detected, affected values in the cache are restored to the state prior to the error.

- Another RMT hardware approach based on a Hardware Transactional Memory (HTM) architecture called LBRA. In this proposal, two redundant threads successfully detect and recover from transient faults, assuring a consistent view of the memory by means of a pair-shared cacheable virtual memory log. The log keeps the computation results and allows a bigger decoupling between redundant threads. This provides better performance than previous approaches while maintaining a relatively simple hardware implementation.

- An analytical model for determining the implications of block-disabling due to random cell failure on cache miss rate behaviour. While previous work rely on simulating a great number of random fault maps to determine the impact of faults in caches, we provide a precise measure for the expected miss ratio and its deviation by applying our analytical off-line model to an application memory trace.

- We have implemented and evaluated the proposed mechanisms as well as previous proposals by using a full system simulator. Therefore, results are presented in a common framework in which comparisons can be easily made. We have found that our mechanisms to provide detection and correction to transient faults improve the performance of previous approaches, at the cost of some extra hardware. Finally, the proposed analytical model for the impact of permanent faults is also validated within this framework.

The contributions of this Thesis have been published in the international peer reviewed conferences and workshops WDDD'08 [96], DPDNS'09 [97], EuroPAR'09 [99], HiPC'10 [98] and IOLTS'11 [100], or are currently being considered for publication in peer reviewed journals.

1.4 Organization of this Thesis

The rest of this Thesis is organized as follows:

- Chapter 2 reviews previous work on fault tolerance in relation with this Thesis. Specifically, this Chapter discusses mechanisms at hardware and software levels as well as hybrid approaches.
- Chapter 3 describes the common methodology used in the evaluation of the different approaches presented in this Thesis. This Chapter discusses the architecture, tools, workloads and metrics used in the evaluation.
- Chapter 4 faces the the major problems presented in previous proposals, i.e. the performance degradation due to the migration to a direct-network environment and several memory consistency issues. Then, our first fault tolerant architecture design is proposed, addressing the aforementioned questions.
- Chapter 5 presents our second fault tolerant approach based on Hardware Transactional Memory. Several mechanisms to reduce the performance penalty and reduce the implementation costs of previous approaches are detailed.
- Chapter 6 addresses the implications of permanent faults in caches. To this end, it is presented the analytical model cited above to evaluate the impact of faults over the cache performance.
- Chapter 7 summarizes the main conclusions of the Thesis and points out future lines of work.

Chapter 2

Background and Related Work

SUMMARY:

In this Chapter we present a summary of previous related work. Specifically, we make an overview of different previous approaches implemented at different hardware levels with special emphasis on architectural-level mechanisms, as well as other software and hybrid approaches.

2.1 Process-Level Measures

As we have seen in Chapter 1, one of the most predominant source of errors comes from the interaction between alpha particles and neutrons with transistor devices. Manufacturers have introduced measures to reduce the SER by radiation-hardened electronics technology. The behaviour of semiconductor devices is highly dependent upon the concentration of impurity elements. Because of that, in the fabrication of semiconductors, extremely high purity materials such as quartz, graphite and other specialized materials like enriched Boron-11 (which is largely immune to radiation damage) are used. Other measures include the separation of high alpha-emission materials from sensitive components and the shielding by

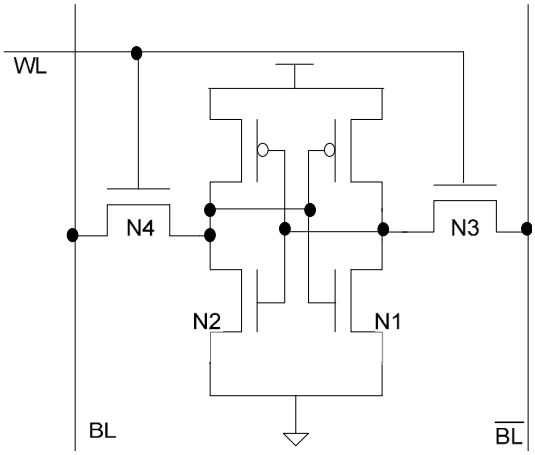
means of polyimide layers [6]. Finally, techniques such as Silicon on Insulator (SOI), in which the substrate is replaced by different insulator-semiconductor-insulator layers, are used to enhance previous techniques.

2.2 Circuit-Level Mechanisms

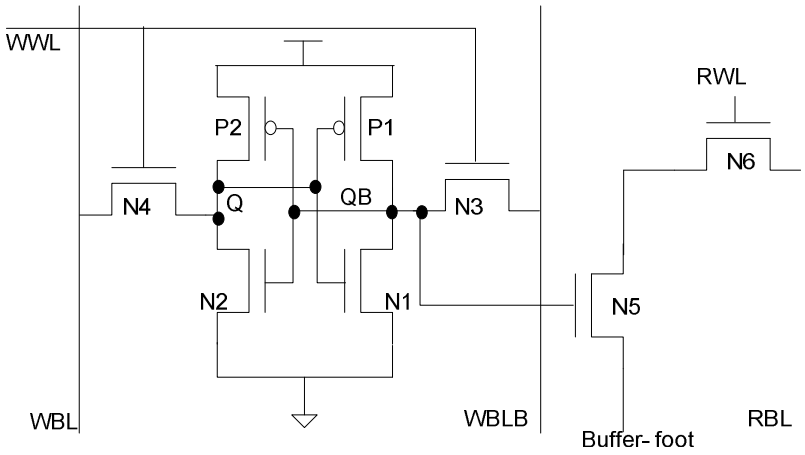
Circuit-level measures usually rely on the use of some form of redundancy. Other mechanisms include special circuits to detect peaks or voltage falls or special codes to detect and/or correct errors.

SRAM cells are commonly implemented with a conventional 6-Transistor structure. This design provides enough stability at normal voltage levels. However, measures to decrease power consumption attempt to reduce the supply voltage. In this case, the SNM (Static Noise Margin) decreases significantly and the arise of soft errors is a fact. Some studies reveal that voltage cannot be scaled down to 0.7 V for an SRAM cell at 65nm to work properly [113, 117]. To mitigate the effects of minimum supply voltage in SRAM cells, engineers have proposed new designs in which more transistors per cell are used. This is the case of 8T [106], in which 2 additional transistors are introduced to obtain higher read SNM, together with 10T [32] designs, which consist of a conventional 6T structure and four decoupled read access transistors. 8T and 10T designs can tolerate voltage scaling to 350mV and 200mV, respectively. In Figure 2.1 we can see the schematic of a 6T, an 8T and a 10T memory cell. Obviously, the major drawback of these designs is the amount of extra hardware that they add. For this reason, the use of these cells is reserved for devices whose reliability is mandatory.

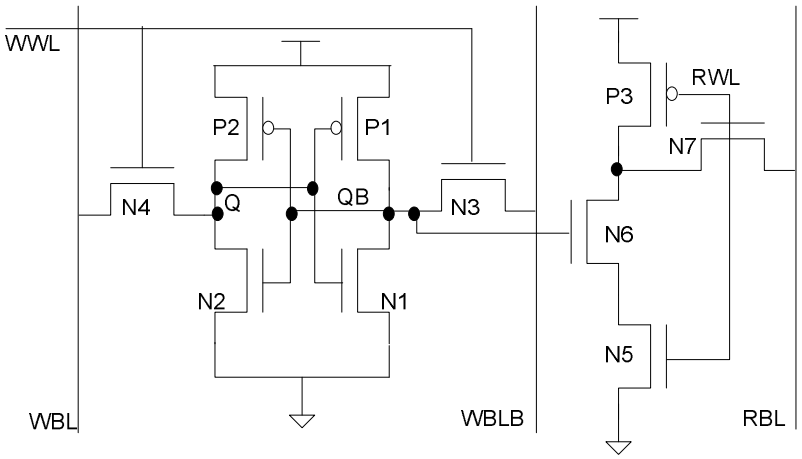
A less aggressive mechanism in terms of hardware requirements is the use of error coding techniques. Parity codes, checksums, Cyclic Redundancy Codes (CRC) and Error Correcting Codes (ECC) fall within this category. When only detection is required, low-cost mechanisms such as parity or checksum are used. Therefore, in the case of a fault, the affected data is invalidated and requested again. However, in other cases in which no other copies of the data exist, error correction is needed. This is the case of memory devices such as caches or main



(a) 6T memory cell [117]



(b) 8T memory cell [106]



(c) 10T memory cell [32]

FIGURE 2.1: Implementation of a memory cell with 6, 8 and 10 transistors.

memory, which must implement, as a consequence, more expensive ECC codes. Typically, implemented ECC codes are able to detect up to 2 bit errors and correct 1 error. We can find implementations in modern processors by manufacturers like IBM [103], Intel [79] and AMD [3]. Nonetheless, ECC is not a power/performance-friendly mechanism given the cost of the decoding (in reads) and coding (in writes). Additionally, ECC is not a good solution for permanently damaged components since every access will incur in the ECC repair overhead.

For the same reasons cited above, parity and ECC are not usually employed to protect the functional units. Instead, several techniques have been proposed to provide error detection and correction to specific units of the architecture. In [38, 50, 105] different self-checking and self-correcting mechanisms for adders based on residue codes and parity prediction are presented. Basically, these mechanisms rely on special properties of the operations they perform (additions in this case) or the use of TMR and temporal redundancy. By using the same principles to protect adders, another body of work [58, 59] addresses fault detection for multipliers.

In general, structures like the register file which holds the architectural state, are more vulnerable to soft errors [54]. In [12], Carretero *et al.* propose several signature-based techniques to protect the rename tables, wake-up logic, select logic, input multiplexors, operand read and writeback, the register free list, and the replay logic with the goal of assuring the correctness of data consumed through registers.

2.3 Architectural-Level Mechanisms

Architectural mechanisms seem a good approach to overcome the major limitations of circuit-level techniques. Previously mentioned mechanisms are able to mitigate the arise of hardware errors but do not provide a full coverage. This is the case of ECC codes, which are limited to detect and correct a few errors. Furthermore, they introduce a fixed hardware overhead which limits, in general, its implementation in real designs. On the contrary, architectural measures provide a more flexible

framework in which multiple hardware structures are covered in comparison to circuit-level techniques which are focused on single units.

One of the most straightforward and studied mechanisms to support fault tolerance is the use of physical and/or time redundancy, in which programs are executed multiple times and compared to detect errors. In Figure 2.2 we can see a conceptual framework for redundant execution.

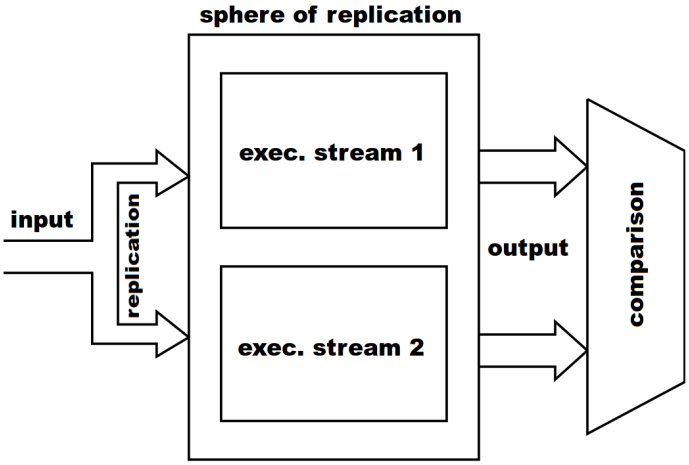


FIGURE 2.2: Redundant execution framework [92].

There are four main characteristics that can be used to classify approaches which rely on redundant execution. These characteristics are the sphere of replication, the input replication, the output comparison and the synchronization.

- **Sphere of replication** or **SoR** [70]. The SoR determines the components in the architecture whose functionality is replicated, i.e., all faults that occur within the sphere will be detected. The size of the sphere of replication could include from whole processors to specific parts of them like the pipelines, include or not the memory, etc...

- **Input replication**. In order to assure that redundant copies perform exactly the same work, they must be provided with the same view of the memory. If not, although redundant executions perform correct computations, they may follow different paths. In tight lock-stepped execution [5], in which redundant streams are in sync and execute the same instructions cycle-by-cycle, input replication is trivial. However, in loose lock-stepped

execution it is necessary the design of specific mechanisms to deal with data races.

- **Output comparison**. The output comparison defines the error detection latency. Generally, a lower latency increases the pressure over the hardware, e.g. comparing the register updates, while a higher latency increases the recovery time after a fault.

- **Synchronization**. Related to the input replication and output comparison is referred the interval of synchronization among redundant computations.

2.3.1 Core-Level Mechanisms

AR-SMT [78] is one of the first proposals which exploits SMT threads for reliability purposes. In AR-SMT, two redundant threads execute the same program instructions. The first thread, or A-thread (Active Thread), runs ahead of the second one, the R-thread (Redundant Thread), by an amount of time determined by the size of a delay buffer. The A-thread pushes the result of register commits and load values in the delay buffer, which are used by the R-thread to compare the correct execution of the program instructions. This way, the A-thread acts like a prefetcher for the R-thread. In case of a fault, the architectural state of the R-stream is used as a safe point and is used to initialize the A-stream to resume execution. AR-SMT, however, requires doubling the physical memory of the system (each stream manage its own memory system), which results very expensive.

Austin proposes the use of a heterogeneous physical redundancy in a CMP with DIVA [4]. DIVA architecture is formed by two different processor units. On the one hand, it has a speculative, superscalar core, the DIVA core. On the other hand, there exists the DIVA checker, a much simpler in-order core with no optimizations which runs slower than the DIVA core but, at the same time, it is almost fault-free. In the absence of faults both cores provide the same results. After verification, instruction results are committed to the architectural state. If a fault is detected, though, the result provided by the DIVA checker is considered as valid, since it

is assumed to be implemented with large and more reliable transistors. However, any fault affecting the checker would pass as undetected in this architecture.

The Slipstream [95] architecture is also based on the execution of redundant threads either in different cores of a CMP or in different contexts in a SMT, providing improved performance and partial fault tolerance. We can see a schematic of the proposed pipeline in Figure 2.3. The A-stream performs a speculative execution of all program instructions and, therefore, in a fastest way than a single equivalent processor. The R-stream, which runs slightly later, executes non-speculatively and checks the correct execution of the A-stream. If a mismatch is detected, the R-stream repairs the architectural state of the A-stream and execution is resumed from that point. The execution of the A-stream is guided by the Instruction Removal predictor (IR-predictor), a component which removes from execution non-effect instructions such as those that write non-consumed values, overwrite the same values or those generating deterministic flows. This mechanism makes the execution of the A-stream go faster. At the same time, the performance of the R-stream is also improved since it is able to use the near-perfect predictions of the A-stream. If a fault affects any of the two streams, this will be reflected as a mismatch between the two computations, something solved by resuming execution from the affected point. Unfortunately, since not all the instructions are executed redundantly, the coverage is partial.

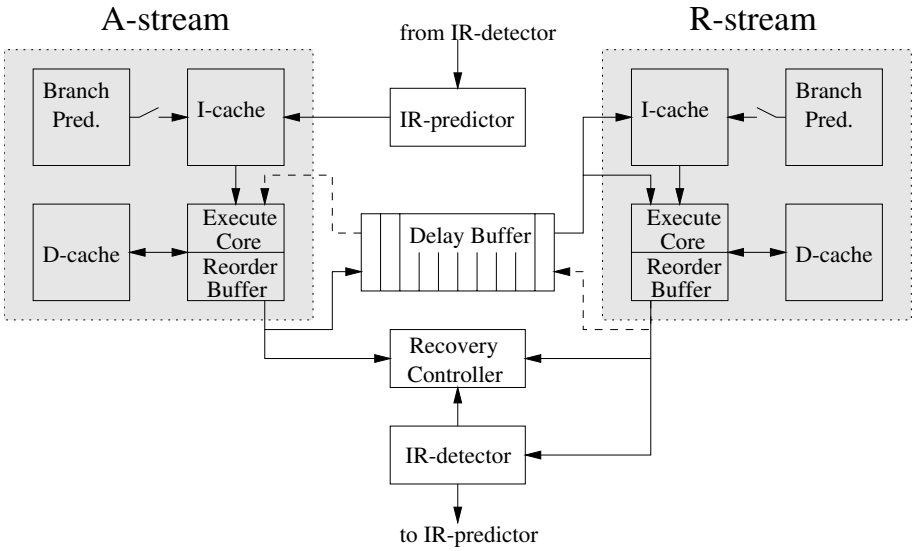


FIGURE 2.3: Slipstream architecture overview [95].

Simultaneous and Redundantly Threaded processors (SRT)[70] and Simultaneous and Redundantly Threaded processors with Recovery (SRTR)[107] implement a fault tolerant design in a SMT processor. To provide input replication between the two redundant threads (leading and trailing thread), SRT proposes two alternative mechanisms, the Active Load Address Buffer (ALAB) and the Load Value Queue (LVQ). Both mechanisms guarantee that corresponding values from redundant threads obtain the same values from the data cache. The ALAB is an associative table which keeps the data blocks read by the leading thread. When a block is about to be replaced or invalidated from the cache, the controller first checks for it in the ALAB. If the block has not been read by the trailing thread yet, the replacement or invalidation for that block is denied. This assures the trailing thread to read the correct value from the cache. The LVQ, however, is much easier to implement. The leading thread keeps committed loaded values and addresses in the LVQ to bypass data to the trailing thread. This way, the data cache is only accessed by loads of the leading thread. Since the LVQ is a FIFO queue, trailing loads need to be performed in program order, which limits, in general, the performance. Additionally, the memory subsystem remains unprotected since load accesses are performed only once. SRTR extends SRT by providing error correction. For that, leading thread commit is stalled until trailing thread check the instructions for faults. Thus, the leading thread is stalled eventually, which increases the performance degradation with respect to SRT.

Gomaa *et al.* [26] improve SRT performance by exploiting partial coverage. They proposed two different execution modes, the Single Execution Mode (SEM) mode, in which leading thread is provided with all processor resources, and the Redundancy Execution Mode (REM), which provides full redundancy. In low ILP phases in which resources are available, REM is activated since the leading thread performance is not affected. In high ILP phases redundancy is switched off to avoid performance degradation. To provide some coverage in SEM, IRTR [85], which exploits dynamic instruction reuse, is proposed.

Slick [63] also reduces SRT degradation by reducing the amount of instructions executed by the trailing thread. For that purpose, it predicts the result of backward slices of verifiable computations (high-confidence branches or easily predictable

stores). In the same fashion, SpecIV [35] makes use of value prediction to avoid trailing thread execution and improve overall performance.

Chip Redundantly Threaded processors (CRT)[55] and Chip Redundantly Threaded processors with Recovery (CRTR)[25] are the adaption to CMP processors of SRT and SRTR, respectively. In these proposals, contrarily to SRT and SRTR, leading and trailing threads are executed within different processor cores. The main benefits obtained are the reduced performance impact because resources are not shared, and the extended coverage, since the same fault cannot affect both executions as a result of spatial distance. However, all the information bypassed between threads increases the inter-core communication traffic. Thus, new wide datapaths are required.

Reunion [83] introduces *relaxed input replication*. The authors rely on the observation that, without additional support, redundant threads which access memory independently, obtain the same memory values most of the time. Thus, instead of relying on strict input replication by means of LVQ or ALAB structures like SRT, they propose to detect divergences between threads and then treat them as the occurrence of transient faults. In Reunion, redundant threads run in different processor cores (as in CRT and CRTR). In order to detect faults, cores frequently interchange *fingerprints* [84], small compressed signatures of the current architectural state. Unfortunately, Reunion requires dedicated point-to-point buses to interchange fingerprints, and the length of checking intervals is reduced to hundred of instructions to avoid excessive penalties because of violations of input replication. Furthermore, these buses must be very fast since Reunion does not commit instructions after comparison. As a consequence, the processor pipeline is stalled frequently, impacting on performance if comparisons are not fast enough.

Dynamic Core Coupling (DCC) [37] avoids the use of special communication channels. For that, DCC increases checking intervals to the order of thousands of cycles while allows instruction commit before checking. The only constraint it imposes is avoiding memory updates to go outside the L1 cache, something which is achieved by adding an unverified bit in memory blocks. When blocks are written in cache, the unverified bit is set and the update of lower levels is not allowed. At the end of the checking interval, if execution is correct, all unverified

bits are cleared. Given the length of these intervals, relaxed input replication is not well suited since it would very frequently lead to violations. Instead, DCC adds a mechanism called consistency window to provide input replication. Unfortunately, this mechanism introduces an undesirable amount of complexity to the coherence protocol. Furthermore, it relies on the use of a shared-bus as interconnection network, which limits the scalability of the design [34]. Alternatively, Rashid *et al.* introduce Highly-Decoupled Thread-Level Redundancy [69] in which the unverified memory updates are buffered in a new structure called Post Commit Buffer (PCB). This way, the consistency window is avoided. Only after checking, PCB buffered values update on-chip caches.

To summarize the above, Table 2.1 shows the main redundant architectures discussed in this Section, and classified according to their SoR, input replication, output comparison and synchronization methods.

TABLE 2.1: Main characteristics of several redundant architectures.

	SoR	**Synchronization**	**Input Replication**	**Output Comparison**
SRT(R) CRT(R)	Pipeline, Registers	Staggered execution	Strict (Queue-based)	Instruction by instruction
Reunion	Pipeline, Registers, L1Cache	Loose coupling	Relaxed input replication	Fingerprints
DCC	Pipeline, Registers, L1Cache	Thousands of instructions	Consistency window	Fingerprints, Checkpoints
HDTLR	Pipeline, Registers, L1Cache	Thousands of instructions	Sub-epochs	Fingerprints, Checkpoints

2.3.2 Coherence-Level Mechanisms

The interconnection network is also prone to transient faults. First because it occupies a significant part of the total chip area, which increases the probability of particles strikes. And second, because it is built with longer wires which exacerbate crosstalking effects.

Sorin *et al.* [87] propose to dynamically verify cache coherence as a measure to detect faults. They introduce end-to-end invariants of the cache coherence protocol and the interconnection network. However, it is limited to snooping coherence protocols and it cannot detect all errors in coherence. Meixner *et al.* [51] extend previous work for any kind of cache coherence protocol, although they do not propose any recovery mechanism. Finally, Fernandez-Pascual *et al.* [21] propose a scheme based on timers set at the start of coherence actions. If a timer expires, it is indicative of an error, something which is usually solved by means of a reply action. This provides the coherence protocol with the ability of recovering itself.

2.4 Software and Hybrid Approaches

Other proposals address fault tolerance at software-level. Although software mechanisms are generally slower, they allow larger flexibility and can be combined with existing solutions in hardware. First approaches like CFCSS [60], EDDI [61] and ACFC [68] rely on the introduction of redundant instructions and asserts that check the correctness of the execution. In the same fashion SWIFT [72] is proposed. SWIFT duplicates all the program instructions but stores. Additionally, it introduces explicit code for checking the control flow.

CRAFT [73], implements RMT in software with hardware support. In contrast to SWIFT, all instructions including loads and stores are duplicated and checked by means of hardware structures, which makes CRAFT a hybrid approach. Original loads access memory and write a new entry in a Load Value Queue (LVQ). Once the redundant load checks that the destination address is the same, both loads are committed. For the treatment of stores the Checking Store Buffer (CSB) is introduced, which keeps the address and value to write. Only when both copies match, the store instruction is issued to memory. Finally, Spot [71] also implements RMT in software with the particularity that allows the user to employ different fault coverage degrees in different parts of the program.

The use of software checkpoints has also been studied in depth since it is relatively easy to deploy in common server hardware. The information kept in the checkpoint includes the core architectural state but also a recent copy for the values of caches and memory. Checkpoints are stored within the memory hierarchy, which reduces the implementation costs and performance overhead. CARER [30] was one the first approaches for uniprocessor systems. CARER holds modified lines in cache until verification. If an error occurs, modified lines are discarded whereas clean lines and memory represent the recovery point. However, in a multiprocessor environment, checkpoints need to be consistent since recovery dependencies may be created as a result of core interactions. Thus, several policies can be taken. The most straightforward one is to take global checkpoints which require complete synchronization. This is the case of proposals such as [23, 50, 53]. Revive [66], additionally performs a memory-based distributed parity protection to recover even from faults affecting the checkpoint. Alternatively, some approaches take coordinated local checkpoints like SafetyNet [89]. In this case, processors take their own checkpoints but eventual interactions with other processors may force to take new checkpoints. This way, independent computations are not forced to sync. Finally, in other approaches cores take uncoordinated local checkpoints [19, 20, 93, 94]. In this case, interactions are recorded. If an error occurs, a register of interactions is used to recover consistently.

2.5 Symptom Based Approaches

Other approaches to fault detection follow a scheme based on symptoms. Symptoms indicate, at different levels, the anomalous behaviour of one or more components of the system. Racunas *et al.* [67] propose several mechanisms to detect data value anomalies caused by faults. These mechanisms are based on invariants or data history values. At microarchitectural level, ReStore [108] detects transient faults based on the observation of abnormal events such as exceptions, cache misses or page faults. Once a fault is detected, the execution is rolled-back to a previous safe state. Finally, Li *et al.* [42] propose to detect faults at software level by capturing events like fatal hardware traps, abnormal application exits, OS hangs,

etc. Unfortunately, this scheme presents several drawbacks. First, faults which do not affect the behaviour but the semantic of applications can go undetected, e.g. faults affecting arithmetic units. And second, the latency of fault detection can be very high since it may take a long time between the occurrence of a fault and its manifestation.

Chapter 3

Simulation Environment and Methodology

SUMMARY:

We have based our studies on the implementation and evaluation of the proposed mechanisms through computer architecture simulators. This brings along the opportunity to parametrize, control and study the targeted system to extract accurate performance metrics whereas maintaining a short development time. Specifically, we have used the Simics-GEMS framework which provides full-system simulation as well as detailed performance analysis.

3.1 Simulation Tools

3.1.1 Simics

Simics [45] is a full-system functional simulator used to run unchanged production binaries of the target hardware at high-performance speeds. Simics can simulate systems such as Alpha, x86-64, IA-64, ARM, MIPS (32- and 64-bit), MSP430, PowerPC (32- and 64-bit), POWER, SPARC-V8 and V9, and x86 CPUs. Many

operating systems can run in Simics including MS-DOS, Windows, VxWorks, OSE, Solaris, FreeBSD, Linux, QNX, and RTEMS. The purpose of simulation in Simics is often to develop software for a particular type of embedded hardware, using Simics as a virtual platform.

3.1.2 GEMS

However, while Simics is able to run any program, it is unable to obtain performance metrics. For that purpose, we have used GEMS [47] simulator, provided by Wisconsin-Madison University. GEMS is a multiprocessor cycle-to-cycle accurate open-source simulator which, coupled to Simics, provides with accurate performance metrics including simulated time, cache hit/miss ratio or network traffic among others. GEMS is formed by different modules leveraging the potential of Virtutech Simics to simulate a Sparc multiprocessor system. Ruby models the memory hierarchy and Opal models an out-of-order speculative pipeline based on the SPARCv9 ISA. Figure 3.1 shows the Simics-GEMS framework. In the elaboration of this thesis we have used Simicsv3.01 and GEMSv2.1.

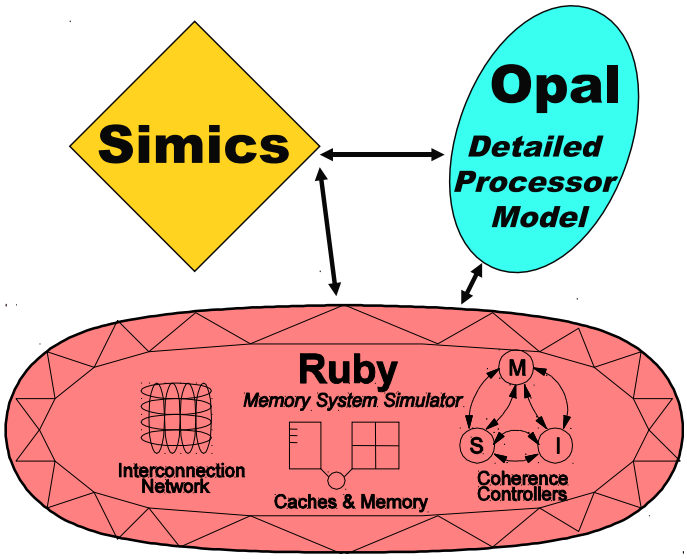


FIGURE 3.1: Simics-GEMS simulator framework.

Ruby implements the interconnection network and the cache/memory subsystem. Additionally, it includes a language to specify cache coherence protocols called SLICC (Specification Language for Implementing Cache Coherence).

SLICC allows us to implement new protocols or manipulate the existing ones by adding new coherence actions and/or states to modify the behaviour of the memory hierarchy. Ruby also models a HTM model based on LogTM [115].

Opal features include a superscalar, pipelined processor core allowing multiple execution units and load/store queues to allow out-of-order memory operations and memory bypassing. The major goal of Opal is to generate multiple memory requests to Ruby. Additionally, it supports the simulation of SMT cores.

3.1.3 McPAT

McPAT (Multicore Power, Area, and Timing) [44] is an integrated power, area, and timing modeling framework for multithreaded, multicore, and manycore architectures. It supports comprehensive early stage design space exploration for multicore and manycore processor configurations ranging from 90nm to 12nm and beyond. McPAT includes models for the components of a complete chip multiprocessor, including in-order and out-of-order processor cores, networks-on-chip, shared caches, and integrated memory controllers. McPAT models timing, area, and dynamic, short-circuit, and leakage power for each of the device types forecast in the ITRS roadmap including bulk CMOS, SOI, and double-gate transistors. McPAT has a flexible XML interface to facilitate its use with different performance simulators.

3.1.3.1 CACTI

CACTI [104] is an integrated cache and memory access time, cycle time, area, leakage, and dynamic power model. By integrating all these models together, users can have confidence that trade-offs between time, power, and area are all based on the same assumptions and, hence, are mutually consistent. CACTI is intended for use by computer architects to better understand the performance trade-offs inherent in memory system organizations. In this Thesis we have used the last version of CACTI, CACTI 6.5 which is included within McPAT v0.8 simulator.

3.2 Simulated System

We use the Simics-GEMS framework to evaluate the performance impact of our proposals in shared-memory CMPs. It is our belief that future many core designs will be organized in tiled-CMPs around a point-to-point network to maintain the scalability that other designs cannot provide [62, 77]. Thus, our base system is formed by a 16-core tiled-CMP organized in a 4x4 2D-mesh. Each tile is formed by a processor core, a private L1 cache and a portion of the shared L2 cache, maintaining a non-inclusive policy between them which provides with a better use of the total available cache capacity. Besides, each tile includes a portion of a distributed directory which is used to provide the coherence of blocks in L1 caches. In Figure 3.2 we can see a diagram of the simulated tiled-CMP architecture.

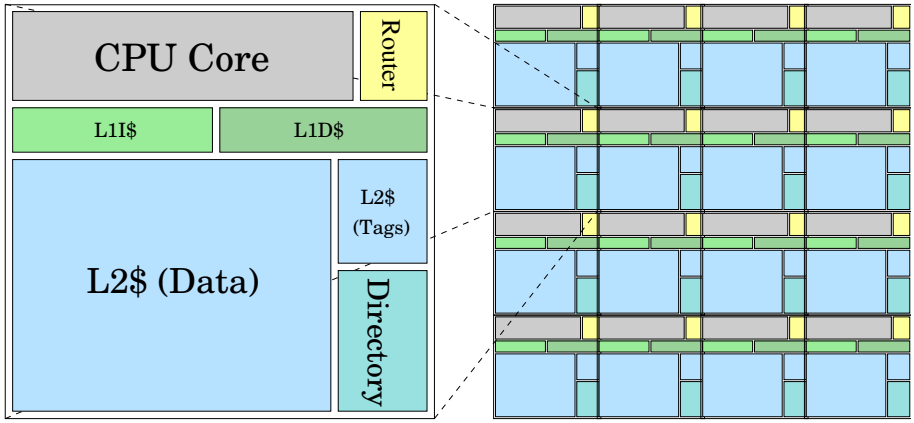


FIGURE 3.2: Organization of a tile and a tiled-CMP system architecture.

3.3 Evaluation Metrics

Basically, we have measured the performance impact of our proposals in terms of execution time. Since the vast majority of evaluated benchmarks are parallel workloads running in a multiprocessor environment, metrics like IPC or CPI are not appropriate as a way to evaluate the performance given the existence of synchronization mechanisms. Instead, we use as main performance metric the number of cycles spent in the execution of the benchmarks.

The synchronization mechanisms are also responsible for the variability of the execution of the evaluated benchmarks, i.e., the results of several simulations are not deterministic (although semantically identical). For that reason, we have taken several runs per benchmark and configuration. Thus, reported results show the average execution time for different runs per benchmark including, additionally, the standard deviation to provide an insight for this variability.

3.4 Benchmarks

The Simics machine emulates a SunFire 6800 server with a UltraSPARC-III processor running Solaris 10 as OS. On top of this system we have evaluated several applications from different benchmark suites ranging from scientific, multimedia and server applications. Specifically, bzip2, gap, gzip, parser, twolf and vpr are sequential benchmarks from the SpecCPU2000 suite [28]. Barnes, cholesky, fft, ocean, radix, raytrace, waternsq and watersp are scientific multithreaded applications from the SPLASH-2 [112] benchmark suite. Unstructured is a fluid dynamic application. Em3d models the propagation of electromagnetic waves through objects. Blackscholes, canneal, swaptions and fluidanimate are applications from the PARSEC [7] benchmark suite. Apache, jbb and tomcatv are server applications. Finally, facerec, MPGdec, MPGenc and speechrec are multimedia applications from the ALPbench [43] benchmark suite.

3.4.1 SpecCPU2000

256.bzip2

Bzip2 compresses files using the Burrows-Wheeler block-sorting text compression algorithm, and Huffman coding. Bzip2 is built on top of libbzip2, a flexible library for handling compressed data in the bzip2 format. The implemented version is based on Julian Seward's bzip2 version 0.1. The only difference between bzip2 0.1 and SpecINT2000 bzip2 is that SPEC's version of bzip2 performs no file I/O other than reading the input. All compression and decompression happens entirely in

memory. This is to help isolate the work done by only the CPU and memory subsystem.

254.gap

Gap is a system for computational discrete algebra, with particular emphasis on computational group theory. It provides a programming language, a library of thousands of functions implementing algebraic algorithms written in the gap language as well as large data libraries of algebraic objects. Gap is used in research and teaching for studying groups and their representations, rings, vector spaces, algebras, combinatorial structures, and more.

164.gzip

Gzip (GNU zip) is a popular data compression program written by Jean-Loup Gailly for the GNU project. It uses Lempel-Ziv coding (LZ77) as its compression algorithm. SPEC's version of gzip performs no file I/O other than reading the input. All compression and decompression happens entirely in memory in order to isolate the work done to by the CPU.

197.parser

The link grammar parser is a syntactic parser of English, based on link grammar, an original theory of English syntax. Given a sentence, the system assigns it a syntactic structure, which consists of a set of labeled links connecting pairs of words. The parser has a dictionary of about 60000 word forms. It has coverage of a wide variety of syntactic constructions, including many rare and idiomatic ones. The parser is able to handle unknown vocabulary and make intelligent guesses from context about the syntactic categories of unknown words.

300.twolf

The TimberWolfSC placement and global routing package is used in the process of creating the lithography artwork needed for the production of microchips. Specifically, it determines the placement and global connections for groups of transistors which constitute the microchip. The placement problem is a permutation. Therefore, a simple or brute force exploration of the state space would take an execution

time proportional to the factorial of the input size. Instead, the TimberWolfSC program uses simulated annealing as a heuristic to find very adequate solutions for the row-based standard cell design style. The simulated annealing algorithm has found the best known solutions to a large group of placement problems. The global router which follows the placement step interconnects the microchip design. It employs a constructive algorithm followed by iterative improvement.

175.vpr

Vpr is a placement and routing program. It automatically implements a technology-mapped circuit (i.e. a netlist, or hypergraph, composed of FPGA logic blocks and I/O pads and their required connections) in a Field-Programmable Gate Array (FPGA) chip. Placement consists of determining which logic block and which I/O pad within the FPGA should implement each of the functions required by the circuit. The goal is to place pieces of logic which are connected (i.e. must communicate) close together in order to minimize the amount of wiring required and to maximize the circuit speed. This is basically a slot assignment problem. Vpr uses simulated annealing to place the circuit. An initial random placement is repeatedly modified through local perturbations in order to increase the quality of the placement, in a method similar to the way metals are slowly cooled to produce strong objects.

3.4.2 SPLASH-2

Barnes

The barnes application simulates the interaction of a system of bodies in 3-D over a number of time steps, using the Barnes-Hut hierarchical N-body method. Each body is modeled as a point mass and exerts forces on all the other bodies in the system. To speed up the interbody force calculations, groups of bodies which are sufficiently far away are abstracted as point masses. In order to facilitate this clustering, physical space is divided recursively, forming an octree. The tree representation of space has to be traversed once for each body and rebuilt after each time step to account for the movement of bodies.

The main data structure in barnes is the tree itself, which is implemented as an array of bodies and space cells which are linked together. Bodies are assigned to processors at the beginning of each time step in a partitioning phase. Each processor calculates the forces exerted on their own subset of bodies. The bodies are then moved under the influence of those forces. Finally, the tree is regenerated for the next time step. There are several barriers for separating different phases of the computation and successive time steps. Some phases require exclusive access to tree cells and a set of distributed locks is used for this purpose. The communication patterns are dependent on the particle distribution and are quite irregular.

Cholesky

The blocked-sparse-cholesky-factorization-kernel factors a sparse matrix into the product of a lower triangular matrix and its transpose. It is similar in structure and partitioning to the LU factorization kernel, but it presents two major differences: first, it operates on sparse matrices, which have a larger communication to computation ratio for comparable problem sizes, and second, it is not globally synchronized between steps.

FFT

The fft kernel is a complex one-dimensional version of the radix$\sqrt{x}$ six-step fft algorithm, which is optimized to minimize interprocessor communication. The dataset consists of the n complex data points to be transformed, and other n complex data points referred to as the roots of unity. Both sets of data are organized as $\sqrt{x}$ x $\sqrt{x}$ partitioned matrices so that every processor is assigned a contiguous set of rows which are allocated in its local memory.

Ocean

The ocean application studies large-scale ocean movements based on eddy and boundary currents. The algorithm simulates a cuboidal basin using discretized circulation model which takes into account wind stress from atmospheric effects and the friction with ocean floor and walls. The algorithm performs the simulation for many time steps until the eddies and mean ocean flow attain a mutual balance. The work performed every time step essentially involves setting up and solving

a set of spatial partial differential equations. For this purpose, the algorithm discretizes the continuous functions by second-order finite-differencing. After that, it sets up the resulting difference equations on two-dimensional fixed-size grids representing horizontal cross-sections of the ocean basin. Finally, it solves these equations using a red-black Gauss-Seidel multigrid equation solver. Each task performs the computational steps on the section of the grids that it owns, regularly communicating with other processes.

Radix

The radix program sorts a series of integers, called keys, using the popular radix sorting method. The algorithm is iterative, performing one iteration for each radix r digit of the keys. In each iteration, a processor passes over its assigned keys and generates a local histogram. The local histograms are then accumulated into a global one. Finally, each processor uses the latter to permute its keys into a new array for the next iteration. This permutation step requires all-to-all communication. The permutation is inherently a sender determined one, so keys are communicated through writes rather than reads.

Raytrace

This application renders a 3-D scene using ray tracing. A hierarchical uniform grid is used to represent the scene and early ray termination is implemented. A ray is traced through each pixel in the image plane and it produces other rays as it strikes the objects of the scene, resulting in a tree of rays per pixel. The image is partitioned among processors in contiguous blocks of pixel groups, and distributed task queues are used with task stealing. The data accesses are highly unpredictable in this application. Synchronization in raytrace is done by using locks. This benchmark is characterized for having very short critical sections and very high contention.

Water-nsq

The water-nsq application performs an N-body molecular dynamics simulation of the forces and potentials in a system of water molecules. It is used to predict some of the physical properties of water in the liquid state. Molecules are statically split

among the processors and the main data structure in water-nsq is a large array of records which is used to store the state of each molecule. At each time step, the processors calculate the interaction of the atoms within each molecule and the interaction of the molecules with one another. For each of them, the owning processor calculates the interactions with only half of the molecules ahead of it in the array. Since the forces between the molecules are symmetric, each pair-wise interaction between them is thus considered only once. The state associated with the molecules is then updated. Although some portions of the molecule state are modified at each interaction, others are only changed between time steps.

Water-sp

This application solves the same problem as water-nsq but using a more efficient algorithm. It imposes a uniform 3-D grid of cells on the problem domain, and uses an $O(n)$ algorithm which is more efficient than water-nsq for large numbers of molecules. The advantage of the grid of cells is that processors which own a cell only need to look at neighboring cells to find molecules that might be within the cutoff radius of molecules in the box it owns. The movement of molecules going in and out of cells causes cell lists to be updated which results in communication.

3.4.3 Parsec 2.1

Blackscholes

This application is an Intel RMS benchmark. It calculates the prices for a portfolio of European options analytically with the Black-Scholes partial differential equation (PDE). There is no closed-form expression for the Black-Scholes equation and, as such, it must be computed numerically. The program divides the portfolio into a number of work units equal to the number of threads and processes them concurrently. Each thread iterates through all derivatives in its contingent and compute the price for each of them.

Canneal

This kernel was developed by Princeton University. It uses cache-aware simulated annealing (SA) to minimize the routing cost of a chip design. Canneal pseudo-randomly picks pairs of elements and tries to swap them. To increase data reuse, the algorithm discards only one element during each iteration which effectively reduces cache capacity misses. Canneal uses a very aggressive synchronization strategy that is based on data race recovery instead of avoidance. Pointers to the elements are dereferenced and swapped atomically, but no locks are held while a potential swap is evaluated. This can cause disadvantageous swaps if one of the relevant elements has been replaced by another thread during that time. This equals a higher effective probability to accept swaps which increase the routing cost, and the SA method automatically recovers from it. The swap operation employs lock-free synchronization which is implemented with atomic instructions.

Fluidanimate

This Intel RMS application uses an extension of the Smoothed Particle Hydrodynamics (SPH) method to simulate an incompressible fluid for interactive animation purposes. The scene geometry employed by fluidanimate is a box in which the fluid resides. All collisions are handled by adding forces in order to change the direction of movement of the involved particles instead of modifying the velocity directly.

Swaptions

The application is an Intel RMS workload which uses the Heath-Jarrow-Morton (HJM) framework to price a portfolio of swaptions. Swaptions employs Monte Carlo (MC) simulation to compute the prices.

3.4.4 Other Scientific Applications

Em3d

Em3d models the propagation of electromagnetic waves through objects in three dimensions. The problem is framed as a computation on a bipartite graph with

directed edges from nodes, representing electric fields to nodes representing magnetic fields and conversely. The sharing patterns found in this application are static and repetitive.

Unstructured

Unstructured is a computational fluid dynamics application that uses an unstructured mesh to model a physical structure, such as an airplane wing or body. The mesh is represented by nodes, edges that connect two nodes, and faces that connect three or four nodes. The mesh is static, so its connectivity does not change. The mesh is partitioned spatially among different processors using a recursive coordinate bisection partitioner. The computation contains a series of loops that iterate over nodes, edges and faces. Most communication occurs along the edges and faces of the mesh.

3.4.5 ALPbench

Facerec

FaceRec is a benchmark based on the Colorado State University face recognizer. Face recognition can be used for authentication, security and screening. Similar algorithms can be used in other image recognition applications. The ALPBench version has been modified to compare a separate input image with all the images contained in a database. The application has an offline training phase, but only the recognition phase has been considered in our evaluation.

MPGdec

The MPGdec benchmark is based on the MSSG MPEG decoder. It decompresses a compressed MPEG-2 bit-stream. Many recent video decoders use similar algorithms. The execution comprises four phases: variable length decoding, inverse quantization, inverse discrete cosine transform (IDCT) and motion compensation. In this application threads are created and finished in a staggered fashion as contiguous rows of blocks are identified by the main thread.We have divided this benchmark in transactions, where each one is the decoding of one video frame.

MPGenc

This benchmark is based on the MSSG MPEG-2 encoder. It converts video frames into a compressed bit-stream. The ALPBench version has been modified to use an intelligent three-step motion search algorithm instead of the original exhaustive search algorithm, and to use a fast integer discrete cosine transform (DCT) butterfly algorithm instead of the original floating point matrix based DCT. Also, the rate control logic has been removed to avoid a serial bottleneck. This application is divided in the same phases as MPGdec, but they are performed in the reverse order. We have divided this benchmark in transactions, where each transaction is the encoding of one video frame. MPGdec and MPGenc perform complementary functions. In fact, we use the output of MPGdec as the input of MPGenc.

Speechrec

The SpeechRec benchmark uses CMU SPHINX speech recognizer version 3.3. Speech recognizers are used with communication, authentication and word processing software and are expected to become a primary component of the human-computer interface. The application has three phases: feature extraction, Gaussian scoring and searching in the language dictionary. The feature extraction phase is not parallelized. Thread barriers are used for synchronization between phases and fine-grain locking is used during the search phase.

3.4.6 Server Applications

Apache

This is a static web serving benchmark using Apache version 2.2.4. Requests are made by a Surge client running on the same machine, simulating 500 clients making random requests to 2000 different HTML files. Each client waits 10ms between requests (think time). The server uses 16 processes and 25 threads per process. It also has all logging functionality disabled. This benchmark is divided in transactions, where each transaction is an HTTP request. We simulate 100,000 transactions without the detailed timing model to warm up the main memory, 500

transactions with the timing model enabled to warm up the caches and 10,000 transactions to obtain results.

Jbb

SpecJbb is based on SPEC JBB2000. It is a Java based server workload that emulates a 3-tier system. In this benchmark, the work of the middle tier, which is the business logic and object manipulation, predominates. We use Sun's HotSpot Java virtual machine version 1.5.0 for Solaris. Our benchmark uses 24 warehouses. This benchmark is divided in transactions. We simulate 1,000,000 transactions without the detailed timing model to warm up main memory, 300 transactions with the timing model enabled to warm up the caches and 8,000 transactions to obtain the results.

Tomcatv

Tomcatv is a parallel version of the SPEC CFP95 - 101.tomcatv benchmark. The program is a vectorized mesh generation program part of Prof. W. Gentzsch's benchmark suite.

Chapter 4

REPAS: Reliable Execution of Parallel ApplicationS in tiled-CMPs

SUMMARY:

In this Chapter we present our first fault-tolerant architecture design which we call REPAS: Reliable Execution of Parallel ApplicationS in tiled-CMPs. With the development of CMPs, the interest in using parallel applications has increased. Previous proposals for providing fault detection and recovery have been mainly based on redundant execution over different cores. RMT (Redundant Multi-Threading) is a family of techniques in which two independent threads, fed with the same inputs, redundantly execute the same instructions in order to detect faults by checking their outputs. In this Chapter, we study the under-explored architectural support of RMT techniques to reliably execute shared-memory applications in tiled-CMPs.

Initially, we show how atomic operations induce serialization points between redundant threads which impacts seriously on execution time. To address this issue, we introduce REPAS, a novel RMT mechanism

to provide reliable execution of shared-memory applications in environments prone to transient faults. REPAS architecture only needs little extra hardware since the redundant execution is performed within 2-way SMT cores in which the majority of hardware is shared. Experimental results show that REPAS is able to provide fault tolerance against soft-errors with a lower execution time overhead than previous proposals in comparison to a non-redundant system, while using less hardware resources. Additionally, we show that REPAS supports huge fault ratios with negligible impact on performance, even for highly unrealistic fault rates.

4.1 Introduction

The advance in the scale of integration allows to increase the number of transistors in a chip, which are used to build powerful processors such as CMPs (Chip Multiprocessors) [102]. But, at the same time, manufacturers have started to notice that this trend, along with voltage reduction and temperature fluctuation, is challenging CMOS technology because of several reliability issues. Among others, we can cite the increasing appearance of hardware errors and other related topics such as process-related cell instability, process variation or in-progress wear-out. Another fact to take into account is that the fault ratio increases due to altitude. Therefore, reliability has become a major design problem in the aerospace industry.

As introduced in Chapter 1, hardware errors are classified as transient, intermittent or permanent [27, 54]. On the one hand, permanent faults, which are usually caused by electromigration, remain in the hardware until the damaged component is replaced. On the other, voltage variation and thermal emergencies are the main cause of intermittent faults.

Transient faults, also known as *soft-errors*, appear and disappear by themselves. They can be induced by a variety of reasons such as transistor variability,

thermal cycling, erratic fluctuations of voltage and radiation external to the chip [54]. Radiation-induced events include alpha-particles from packaging materials and neutrons from atmosphere. As we have shown in Chapter 1, it is well established that the charge of an alpha particle or a neutron strike over a logical device can overwhelm the circuit inducing its malfunction.

It is hard to find documented cases concerning soft errors in commercial systems. This is as result of both the difficulty which involves detecting a soft error and the convenient silence of manufacturers about their reliability problems. However, several studies show how soft errors can heavily damage industry. For instance, in 1984 Intel had certain problems delivering chips to AT&T as a result of alpha particle contamination in the manufacturing process [54]. In 2000, a reliability problem was reported by Sun Microsystems in its UltraSparc-II servers deriving from insufficient protection in the SRAM [54]. A report from Cypress Semiconductor showed how a car factory had to be halted once a month because of soft errors [119].

Nowadays, several measures have been introduced in microarchitectural designs in order to detect and recover from transient errors such as error detection and correction codes. They are created by specific rules of construction to avoid information loss in the transmission of data. ECC codes are commonly used in dynamic RAM. However, these mechanisms cannot be extensively used across all hardware structures. Instead, at the architecture level, DMR (Dual Modular Redundancy) or TMR (Triple Modular Redundancy) have been proposed. In these approaches, fault detection is provided by means of dual and triple execution redundancy.

In this fashion, we find RMT (Redundant Multi-Threading), a family of techniques in which two threads redundantly execute the program instructions. Simultaneous and Redundantly Threaded processors (SRT) [70] and SRT with Recovery (SRTR) [107] are two of them (see Section 2.3.1), implemented over SMT processors in which two independent and redundant threads are executed with a delay respect to the other which speeds up their execution. These early approaches are attractive since they do not require many design changes in a traditional SMT processor. In addition, they only add some extra hardware for communication

purposes between the threads. However, the major drawback of SRT(R) is the inherent non-scalability of SMT processors as the number of threads increases.

In order to provide more scalability, several approaches were designed on top of CMP architectures, such as Reunion [83], Dynamic Core Coupling (DCC) [37] or High Decoupled Thread Level Redundancy (HDTLR) [69], as cited in Chapter 2. However, solutions using this kind of redundancy achieve a severe degradation in terms of power, performance and especially in area, since they use twice the number of cores to support DMR. Therefore, these approaches are not well suited for general markets, as the industry claims that a fault tolerant mechanism should not impose more than 10% of area overhead in order to be effectively deployed [74]. Hence, solutions based on redundant multithreading using SMT cores seem a good approach to achieve fault tolerance without sacrificing too much hardware [42].

Although there are different proposals based on SRTR with either sequential or independent multithreaded applications [107][35], the architectural support for redundant execution with shared-memory workloads is not well suited. As we will show in Section 4.2.2, in shared-memory parallel applications, the use of atomic operations may induce serialization points between master and slave threads, affecting performance in a grade which depends on the memory consistency model provided by the hardware.

To address all these issues, in this Chapter we propose REPAS, *Reliable Execution of Parallel ApplicationS* in tiled-CMPs. The main contributions of this work are: a) a performance problem of traditional RMT implementations has been identified; b) to address this issue, a scalable RMT solution built on top of dual SMT cores to form a reliable CMP has been designed; and c) a proposal has been implemented in a full-system simulator to measure its effectiveness and execution time overhead. We show that REPAS is able to reduce the execution time overhead down to 25% with respect to a non fault-tolerant architecture, while outperforming a traditional RMT mechanism by 13%. Previous proposals, such as DCC, obtain a better performance for specific environments such as Multimedia and Web Server applications. However, REPAS achieves the same goal by using half the hardware

(cores) used in DCC. Additionally, our mechanism is able to recover from transient faults with negligible performance impact even with extremely high fault rates.

The rest of the Chapter is organized as follows: Section 4.2 reviews some related work. In Section 4.2.1 we further detail DCC, a previous related fault tolerant mechanism, for comparison purposes. Section 4.2.2 introduces CRTR and presents its major drawbacks in a parallel shared-memory environment. We present REPAS's architecture in Section 4.3. Section 4.4 analyzes the performance of REPAS in both fault-free and faulty environments. Finally, Section 4.5 summarizes the main conclusions of this work.

4.2 RMT Previous Approaches

As mentioned before, in this Chapter we compare REPAS with two state-of-the-art related proposals aimed at providing fault tolerant execution in a CMP environment: Chip Redundantly Threaded processors with Recovery (CRTR) [25] and Dynamic Core Coupling (DCC) [37]. Although both proposals were briefly sketched in Chapter 2, this Section shows a more extensive and detailed explanation of their peculiarities in order to provide a more comprehensive view of each one so as to show their benefits and drawbacks.

4.2.1 Moving Dynamic Core Coupling to a Direct Network Environment

Dynamic Core Coupling (DCC) [37] is a fault tolerant mechanism for both sequential and parallel applications. DCC implements dual modular redundancy (DMR) by binding pairs of cores in a CMP connected by a shared-bus. To provide fault tolerance, paired cores redundantly execute program instructions to verify each other's execution. In this Section, we deeply analyze the major benefits and drawbacks of DCC. In particular, we focus on the impact over the coherence and consistency systems of DCC when it is ported from a shared-bus to a more scalable direct-network.

4.2.1.1 DCC in a Shared-Bus Scenario

In DCC, a pair is formed by two cores, the master core and the slave core, which redundantly execute all the program instructions. To verify the correct execution, at the end of a checkpoint interval each master-slave pair interchange the compressed state of their register file and all the updates performed to memory. In order to amortize the compression time and save bandwidth these checkpoints intervals are in the order of 10,000 cycles.

Both, master and slave cores are allowed to read memory. However, only the master is permitted to modify and share memory values. Writes to memory are marked in L1 cache by means of an *unverified bit* [48]. This bit indicates that the modification of the block has not been verified yet. In order to avoid the propagation of errors, the update of unverified values in lower levels of the memory hierarchy (L2 and beyond) is not allowed. At the end of each checkpoint interval all the unverified bits are cleared.

To provide a correct execution of parallel applications, DCC needs several changes to both the coherence and consistency system. As said before, the master core is the responsible of sharing unverified data. From the point of view of coherence, this means that the slave core is not allowed to response to forwarded requests (request from other cores), although invalidations must be performed accordingly by evicting blocks from cache (without updating lower levels of the memory hierarchy). The constraints that DCC should address to provide a correct execution increase noticeably the complexity of the coherence protocol. These constraints include:

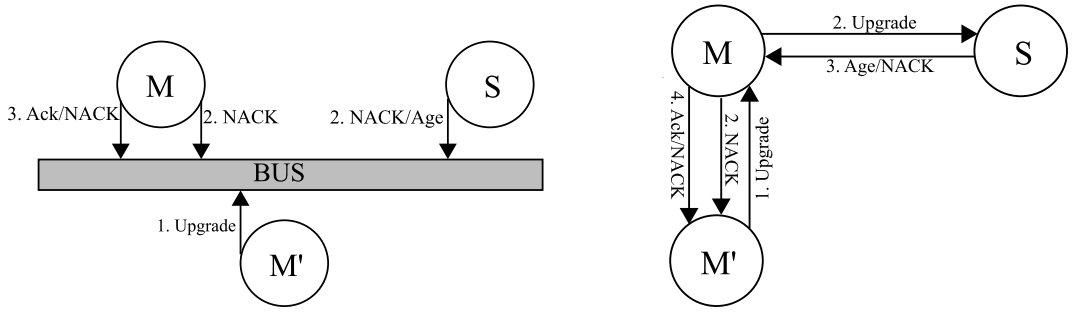
- To assure forward progress, writes to verified dirty cache blocks force a write-back to L2 in master cores. Contrarily, slave cores never update L2.
- A reader marks its line as unverified if in the original holder the line is unverified.
- Slave cores never supply data on a request to a remote core[1].

[1] A remote core is any core different from its master pair.

- The coherence protocol should be MOESI (or similar), to provide datablock sharing among nodes without updates in memory.
- Slave reads could downgrade block states by accesses in remote caches but never could cause invalidations.
- Master reads to unverified lines could cause invalidations in a remote master, only in the case that their slave-pairs keep a copy of the line, which is marked as unverified to prevent eviction. If the slave has no copy of the line, by using the capability offered by the shared bus, it will read the message directed to the remote core to get the line. The same happens with master upgrades.
- The replacement of unverified blocks in L1 cache causes a buffering overflow. To avoid the propagation of unverified data a new checkpoint must be created.

The major difficulty, however, is to provide the master-slave consistency to assure than both cores obtain the same view of the memory at all times. The pair consistency is violated if between the time a master's read is performed an intervening write modifies the value, preventing the slave's (redundant) read from obtaining the same value as the first one. This problem is solved in DCC by a set of constraints referred to as the *master-slave consistency window*. Logically, a consistency window represents a time interval in which any remote intervention could cause a violation of the consistency. For example, a consistency read window is open on any master read and is closed once the slave core commits the same read. To avoid consistency violations, it must be assured that no write windows are opened for an address in which another window has been previously open.

DCC implements this mechanism by means of an age table. The age table keeps, for every load and store, the number of committed loads and stores since the last checkpoint. In Figure 4.1(a) we can see how this mechanism works. A node requests an upgrade or a read-exclusive for a block through the shared-bus (the request is seen by all nodes) (1). Each core checks its LSQ in case a speculative load has been issued. If this is the case, the request in NACKed (2). Parallelly, each core accesses its age table and reports it to its pair (2). In the following



(a) Shared-bus (b) Direct-network

FIGURE 4.1: DCC master-slave consistency.

cycle, every master core checks its own age with the slave's one. In case of a mismatch, it means that a window is open and, therefore, the request is NACKed to avoid a master-slave inconsistency (3). If no mismatch is found the request can be satisfied.

4.2.1.2 DCC in a Direct-Network Scenario

As the number of cores in a system grows, we observe undesirable effects impacting on the scalability of systems. One of the affected elements is the interconnection network. As shown in [34], the area required by a shared-bus (or a crossbar) increases to the point of becoming impractical as the number of cores grows. Therefore, one of the first works we perform in this Thesis is the analysis and evaluation of moving DCC towards a point-to-point unordered network, a more scalable alternative for CMP designs.

In order to accommodate the behaviour of DCC to a direct-network, we should introduce several changes in both the coherence and consistency mechanisms. In both cases the problem is the same: without additional support, slave cores are unaware of coherence actions because of the loss of the shared-bus and its "broadcast" capabilities. We solve this issue by redirecting coherence messages (upgrade, read-exclusive and invalidation requests) which arrive to master cores to their slave pairs, introducing, unfortunately, additional traffic and a delay in the communication.

We can see how this problem affects the way in which the consistency window works in Figure 4.1(b). Upgrade, read-exclusive or invalidations are now sent to master cores which are the visible cores in the system (1). These requests can be directly NACKed in case a speculative load is performed in the master (2). In parallel, the request needs to be sent to the slave core which, so far, was unaware of the coherence action. The slave core can deny the request through the master core in case a speculative load is found in its LSQ (3). Otherwise, it sends its age to its master pair. Finally, the master core checks for a window violation and then informs the requestor (4). As we can see, an additional hop is introduced in the communication for every coherence action.

As we can see in Figure 4.2, there is another problem related to inconsistencies when replacing verified data blocks. After the replacement of block A in $Master_1$, another core, $Master_2$, acquires the block and eventually modifies it. When $Slave_1$ executes the redundant load, it will perceive a different value. In the original DCC proposal with a shared-bus, the consistency window is able to resolve this conflict. In spite of having the block replaced, $Master_1$ and/or $Slave_1$ can see through the bus that an external core ($Master_2$) has issued a read-exclusive or an upgrade request, therefore, aborting the request. However, in a direct network, the redundant pair is unaware of the fact that another core wants to acquire the block for writing purposes, since there is no information to guide the message from the requestor ($Master_2$) to the previous owner which replaced the block.

If we would like to imitate the shared-bus DCC behaviour to avoid these consistency errors, requests should be flooded all over the network. This solution, however, creates a large amount of network traffic with a big latency. Hence, we propose a simpler solution which consists of extending the consistency window concept to cache replacements. For that purpose, on every replacement, the leading core must check that its pair has read the block. If the partner possesses the block, the replacement can be executed. If not, it will be delayed until the pair reads the block some cycles later, causing a necessary latency overhead in L1 replacements. In this way, potential consistency errors between masters and slaves will be solved.

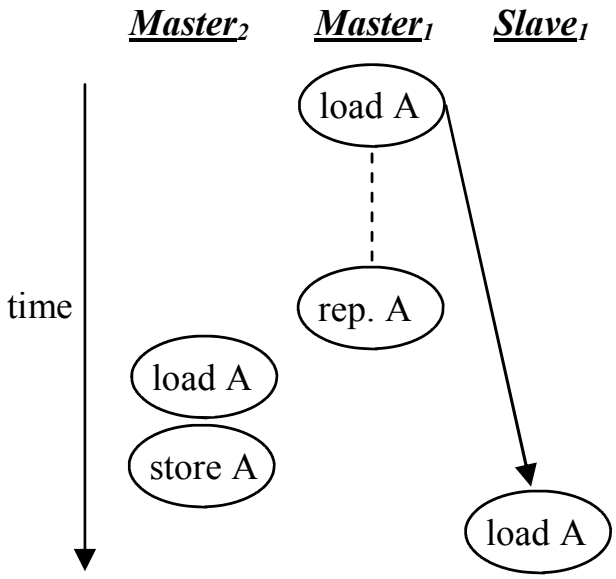


FIGURE 4.2: Potential consistency error in DCC.

Finally, another relevant aspect when using a direct network to fit the original DCC proposal is synchronization when creating checkpoints, as illustrated in Figure 4.3. The synchronization request is issued at the end of a scheduled interval or when events such as buffering overflows occur. In a direct network, the responsible for sending a synchronization request is called *Initiator*. The *Initiator* has to send a message to every master in the system. When the request is received, each master tries to sync with its slave-pair, creating and exchanging its architectural state using a fingerprint [84]. If fingerprints match with each other, an acknowledgment is sent back to the *Initiator*. Once all the acknowledgements have been received, the *Initiator* finally sends a message to each core in the system, giving the order to save the current state as the last checkpoint and, finally, all cores resume execution. Besides, if one core finds a mismatch when comparing fingerprints, a NACK indicating a transient fault detection will be sent to the *Initiator*, which will expand the information, causing every core in the system to rollback to its previously saved checkpoint.

This mechanism for creating new checkpoints exhibits a variable latency directly dependent on the distance and the network congestion between the *Initiator* and the furthermost core. Thus, the *Initiator* will not send the save-state request until all ACKs confirming the synchronization have arrived. If any message could

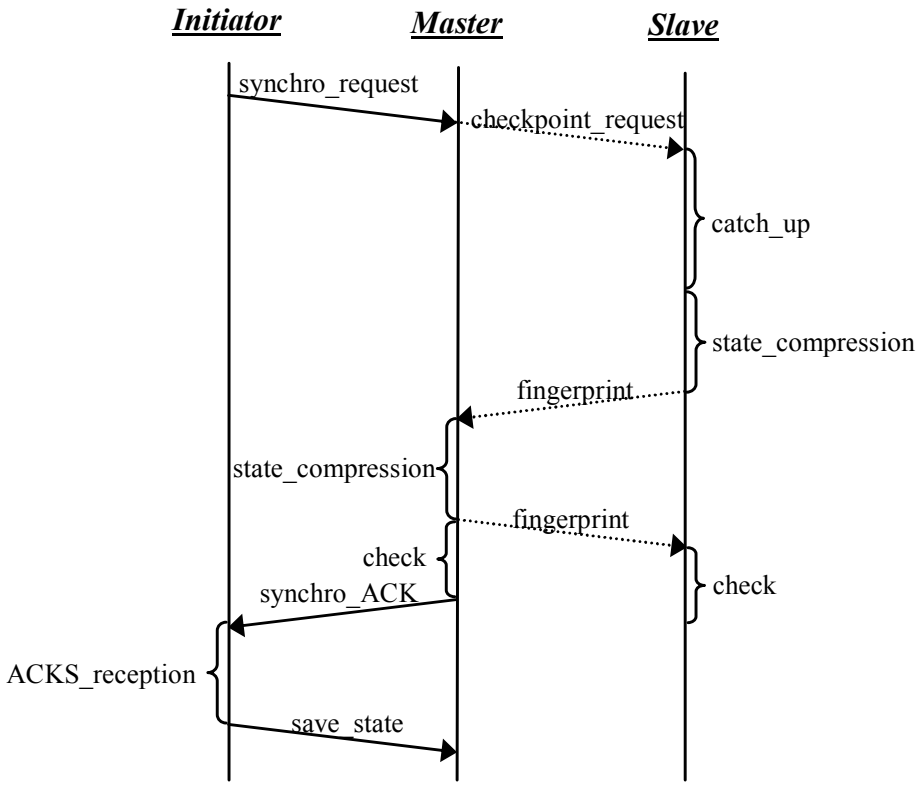


FIGURE 4.3: Synchronization and checkpoint creation.

not arrive due to a permanent fault in one core, the *Initiator* would be waiting in an infinite loop. To avoid this situation, a timeout is set up when waiting for ACKs. The impact of these measures over the performance is studied and analyzed in Section 4.4.3.

4.2.2 CRTR as a Building Block for Reliability

As opposed to DCC in which the redundancy is taken by using master-slave pairs in different cores (increasing the hardware overhead), another alternative consists of the use of dual SMT cores. This way we can reduce both the hardware overhead and the delay caused by the communications between redundant pairs through the network. Among different alternatives using SMT cores we focus on Chip-level Redundantly Threaded multiprocessors with Recovery (CRTR).

CRTR is a fault-tolerant architecture proposed by Gomaa *et al.* [25], an extension of SRTR [107] for CMP environments. In CRTR, two redundant threads

are executed on separate SMT processor cores, providing transient fault detection. These threads are called *master* (or *leading*) and *slave* (or *trailing*) threads, since one of them runs ahead the other by a number of instructions determined by the *slack*. As in a traditional SMT processor, each thread owns a PC register, a renaming map table and a register file, whereas all the other resources are shared.

In CRTR, the master thread is responsible for accessing memory to load data. After a master load commits, it bypasses it to the slave thread along with the accessed address through a FIFO structure called Load Value Queue (LVQ) [70]. This structure is accessed by the slave thread, avoiding to observe different values from those the master did, a phenomenon called *input incoherence*. To avoid associative searches in the LVQ, the slave thread executes loads in program order so it only has to lookup the head of the queue. Fortunately, this handicap does not impact on the slave's performance in comparison to the master's, because the possible slowdown is compensated with a speedup resulting from two factors:

- The memory latency of a slave load is very low since data is provided by the LVQ (slave's loads behave as cache hits).
- Branch mispredictions are completely avoided thanks to the Branch Outcome Queue (BOQ) [70]. Therefore, the slave thread executes less instructions than the master.

The master uses the BOQ to bypass the outcome of a committed branch. Then, the slave accesses the BOQ at a branch execution obtaining accurate predictions (perfect outcomes, in fact). Availability for these hints is assured thanks to the slack since, by the time the slave needs to predict a branch, the master has already logged its correct destination of the branch in the BOQ.

To avoid data corruptions, CRTR never updates cache before values are verified. To accomplish this, when a store instruction is committed by the master, the value and accessed address are bypassed to the slave through a structure called Store Value Queue (SVQ) [70]. When a store commits in the slave, it verifies the SVQ and, if the check succeeds, the L1 cache is updated. Finally, another structure used in CRTR is the Register Value Queue (RVQ) [107]. The RVQ is used

to bypass register values of every committed instruction by the master, which are needed for checking correctness.

Whenever a fault is detected, the recovery mechanism is triggered. The slave register file is a safe point since no updates are performed on it until a successful verification. Therefore, the slave bypasses the contents of its register file to the master, pipelines of both threads are flushed and, finally, execution is restarted from the detected faulty instruction.

As commented before, separating the execution of a master thread and its corresponding slave in different physical SMT cores adds the ability to tolerate permanent faults. However, it requires a wide datapath between cores in order to bypass all the information required for checking. Furthermore, although wire delays may be hidden by the slack, cores exchanging data must be close to each other to avoid stalling.

4.2.2.1 Memory Consistency in LVQ-Based Architectures

Although CRTR was originally evaluated with sequential applications [25, 55], the authors argue that it could be used for multithreaded applications, too. In LVQ-based systems such as CRTR in which loads are performed by the master thread and stores are performed by the slave thread, there is a significant reordering in the memory instructions from an external perspective. In a sequential environment, it does not represent any problem. However, for shared-memory workloads in a CMP scenario, though, if no additional measures are taken, CRTR can lead to severe performance degradation due to consistency model constraints.

Our evaluated architecture is a SPARC V9 [109] implementing the Total Store Order (TSO) consistency model. In this consistency model, stores are buffered on a store miss while loads are allowed to bypass them. As a measure to improve the performance, stores to the same cache block are coalesced in the store buffer. Finally, atomic instructions and memory fences stall retirement until the store buffer is drained.

In shared-memory applications such as those which can be found in either scientific SPLASH-2 [112] or multimedia workloads, the access to critical sections is granted by acquisition primitives relying on atomic instructions and memory fences. We have noticed that, in this environment, CRTR could lead to performance loss because of the constraints of the consistency model to assure mutual exclusion.

The key point is that, in CRTR, memory is never updated by the master thread. Therefore, when a master executes the code to access a critical section, the acquisition is not made visible until the slave executes and verifies the correctness of the instructions involved. This means that, for the rest of master threads, the *lock* remains free for an undetermined period of time, enabling two (or more) of these threads to access a critical section as illustrated in Figure 4.4. In the Figure, two master threads M_0 and M_1, and one slave thread S_0 are presented (the corresponding slave for M_1 has been omitted for simplicity). Part 4.4(a) shows a snapshot of the program execution. M_0 runs ahead of S_0 by an amount of instructions determined by the slack. A stripped portion of a bar means that updates to memory have not been performed yet. Part 4.4(b) shows the situation when M_0 acquires a lock and enters the critical section it protects. None of the modifications are visible yet. Part 4.4(c) shows that M_1 also acquires the lock some cycles later. This is because M_0 has not updated memory so the lock seems free for the rest of the nodes in the system. M_1 enters the critical section at the same time as M_0. Finally, part 4.4(d) shows that when S_0 validates the execution of M_0 and updates memory values, it is too late, since atomicity and isolation of the critical section have been violated.

To address this issue which appears in CRTR without modifying the memory consistency model, we propose to implement and evaluate two different alternatives: *atomic synchronization* and *atomic speculation*.

CRTR with Atomic Synchronization

In order to preserve the underlying consistency model (TSO) and, therefore, the correct program execution, the most straightforward solution is to synchronize the master and slave threads whenever atomic instructions or memory fences are

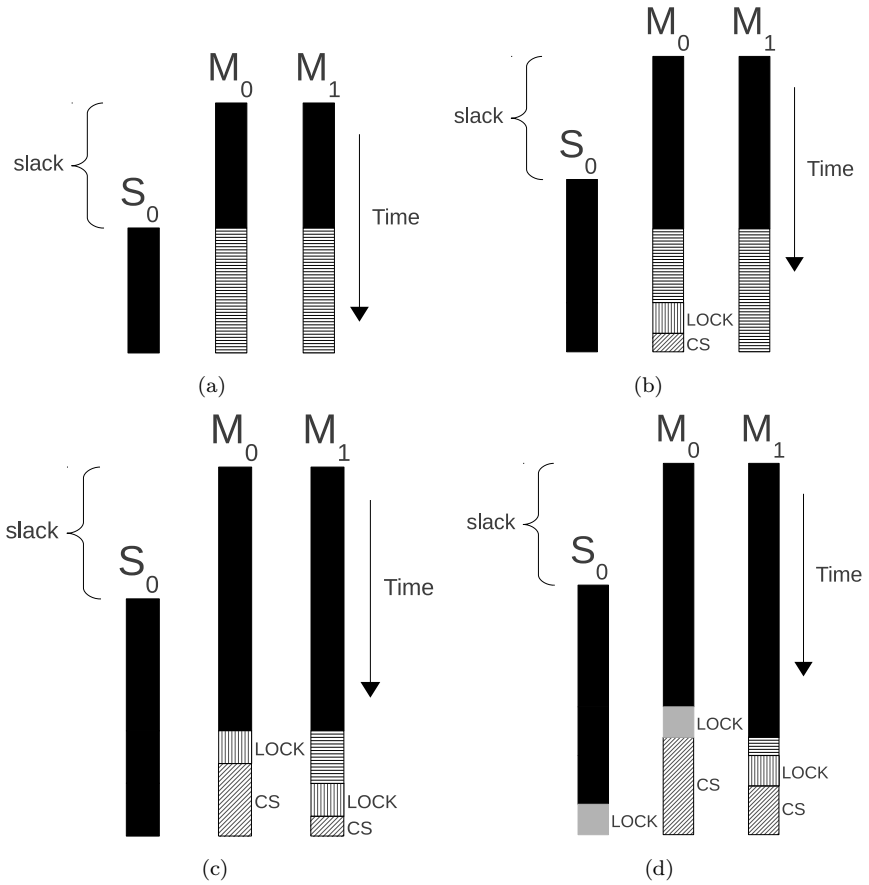


FIGURE 4.4: Violation of the atomicity and isolation of a critical section without proper support.

executed. This way, only when the slave thread catches up with the master and the SVQ drains, the instruction is issued to memory. Therefore, the master thread is not allowed to enter into a critical section without making the results of the acquisition mechanism visible.

Note that this is a conservative approach which introduces a noticeable performance degradation because the retirement on every atomic/memory fence instruction is stalled by the master. The duration of this stall depends on two factors: (1) the size of the slack, which determines how far the slave thread is; and (2) the number of write operations in the SVQ, which must be written in L1 prior to the atomic operation to preserve consistency.

CRTR with Atomic Speculation

One could argue that the previous alternative is not fair to competition. To this end, we have evaluated a mechanism, which we called *Atomic Speculation* to relax even more the consistency constraints imposed by TSO through the use of speculation. Memory ordering speculation has been previously studied in [8, 24, 110] in order to increase the performance of different consistency models.

What we try to accomplish with Atomic Speculation is to avoid the costly synchronizations that atomic instructions and memory fences impose over CRTR. For this, we allow loads and stores to bypass these instructions speculatively. In the same fashion as in [24], the list of speculated blocks is maintained in a hardware structure in the core[2]. A hit in the table upon a coherence message from other core indicates that the current speculation could potentially lead to a consistency violation. In this situation, a conflict manager decides whether to roll-back the receiver or the requestor because of the miss-speculation. Eventually, if no violations have been detected, the slave thread will catch up with the master. Then, the speculation table is flushed and the speculative mode is finished.

In benchmarks with low to medium synchronization time this kind of speculative mechanism results in a good approach. However in other scenarios with highly contended locks the frequency of rollbacks severely impacts on performance. Nonetheless, this mechanism comes at an additional cost, such as the hardware needed to rollback the architecture upon a consistency violation. Additionally, this solution requires a change in the way atomicity is implemented since these accesses cannot perform the memory update to avoid fault propagation. Finally, there exists a power consumption overhead due to the need of checking the speculation table for every coherence request in the speculative mode. Note, however, that we have not considered these overheads in the evaluation of this approach.

[2]The same goal could be accomplished by means of signatures as in certain hardware approaches of Hardware Transactional Memory.

4.3 REPAS Architecture

At this point, we present *Reliable Execution for Parallel ApplicationS in tiled-CMPs* (REPAS) [99]. We create the reliable architecture of REPAS by adding CRTR cores to form a tiled-CMP. However, instead of separating master and slave threads in different cores, we rely on the use of 2-way SMT cores. This way, the architecture avoids the use of the expensive inter-core datapaths whereas still offers fault tolerance to soft errors and adequate performance.

An overview of the core architecture is depicted in Figure 4.5. As in a traditional SMT processor, issue queues, register file, functional units and L1-cache are shared among the master and slave threads. The shaded boxes in Figure 4.5 represent the extra hardware introduced by CRTR and REPAS as explained in Section 4.2.2.

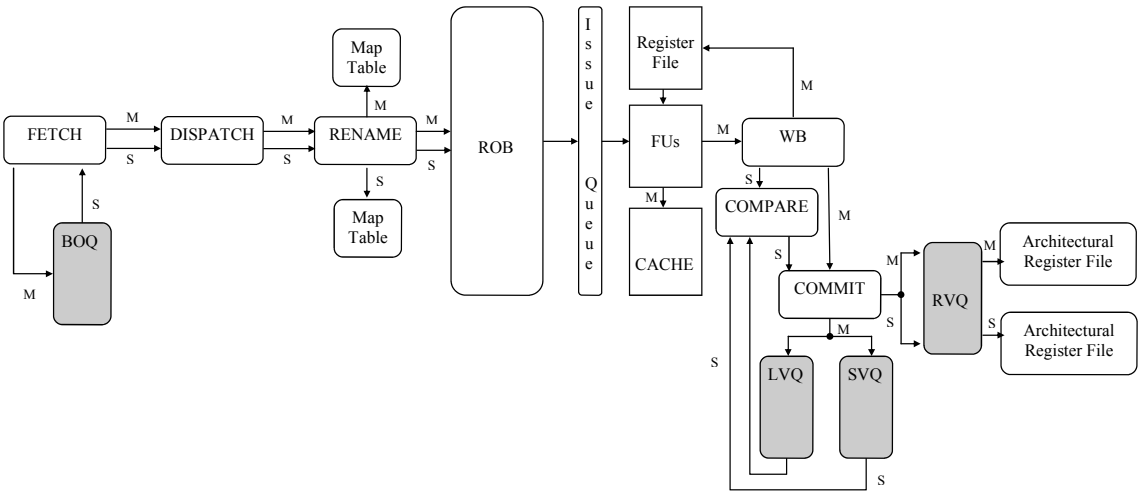


FIGURE 4.5: REPAS core architecture overview.

4.3.1 Sphere of Replication in REPAS

In benchmarks with high contention resulting from synchronization, the approaches described in Section 4.2.2 for CRTR may increase the performance degradation of the architecture due to atomic synchronizations or too frequent rollbacks because of miss-speculations. To avoid frequent master stalls derived from consistency, we

propose an alternative management of stores in REPAS. Instead of updating memory only after verification, a more suitable approach is to allow updates in L1 cache without checking. This measure implies that unverified data could go outside the SoR while the master thread will not be stalled as a result of synchronizations.

Additionally, with this new behaviour we effectively reduce the pressure on the SVQ queue. In the original CRTR implementation, a master's load must look into the SVQ to obtain the value produced by an earlier store. This implies an associative search along the structure for every load instruction. In REPAS, we eliminate these searches since the up-to-date values for every block are stored in L1 cache where they can be accessed as usual.

However, this change in the SoR with regards to CRTR entails an increase in the complexity of the recovery mechanism and the management of verified data. In our approach, in contrast to CRTR, when a fault is detected, the L1 cache may have unverified blocks. The recovery mechanism involves the invalidation of all the unverified blocks in L1. In order to maintain L2 updated with the most up-to-date versions of blocks, when stores are correctly checked by the slave, the values in the SVQ must be written-back into L2. This way, the L2 cache remains consistent even if the block in L1 is invalidated as a result of the mechanism triggered because of a fault. To perform these writebacks we use a small coalescing buffer to mitigate the increase of the SVQ-to-L2 traffic in the same fashion as in [69]. Despite the increasing SVQ-to-L2 traffic, there is no noticeable impact on performance.

4.3.2 Caching Unverified Blocks

To avoid error propagation in REPAS as a consequence of an incorrect result stored in L1 cache by the master, unverified blocks in cache must be identified. To accomplish this, we make use of an additional bit per L1 cache block called *Unverified bit* which is activated on any master write (as already proposed by DCC [37]). In our proposed REPAS, when the Unverified bit is set on a cache block, it cannot be displaced or shared with other nodes, effectively avoiding the propagation of a faulty block. Eventually, the Unverified bit will be cleared when the corresponding slave thread verifies the correct execution of the memory update.

This mechanism is controlled at the coherence protocol level by adding a new state (M_Unv) to the base MOESI protocol as we can see in Figure 4.6. Modified blocks remain in M_Unv state until a positive verification is performed by the slave. Upon this verification, the state of the block transitions from M_Unv to M state where it can be shared or replaced normally.

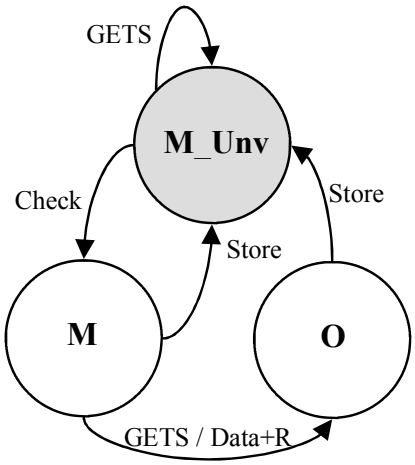


FIGURE 4.6: Transition diagram with the states involved with Unverified blocks.

However, clearing the Unverified bit is not a trivial task. We might find a problem when a master thread updates a cache block several times before verification takes place. If the first check performed by the slave is successful, it means that the first memory update was valid. However, this does not imply that the whole block is completely verified since the rest of the updates have not been checked yet. We propose two different mechanisms in order to address this issue.

The first mechanism is based on counters per L1-cache block. Each time that the master thread updates a block it increments the counter which is eventually decremented when a verification is performed. When the counter is 0, the state of the blocks moves from *M_Unv* to *M*, meaning that the block has been successfully verified. However, these counters have a deep impact in hardware overhead. With small 4-bit counters (which can only record up to 15 consecutive updates) the area overhead becomes around 6% with a 64KB L1 cache and 64-byte blocks.

Thus, we have adopted a more lightweight mechanism based on the following observation regarding the SVQ: we also know that a block needs more slave checks before clearing the unverified bit by checking if the block appears more than once in the SVQ. If it does, more verifications need to be performed. Yet, this measure implies an associative search in the SVQ. Nonetheless, as we said before, we eliminate much of the pressure produced by master's loads. In quantitative terms, in the original CRTR proposal there was an associative search every master's load, but now in REPAS we have an associative search every slave's store. This results in a significant reduction of associative searches within the SVQ, given the fact that the load/store ratio for the studied benchmarks is almost 3 to 1. Furthermore, as this operation is performed in parallel to the access to L1 cache, we do not expect an increase in the L1-cache access latency.

4.3.3 Fetch and ROB Occupancy Policies

The most common fetch policy for SMT processors is *round-robin* in which each thread fetches instructions in alternative cycles. In REPAS, the fetch policy needs to interact with the slack mechanism, which significantly differs from the requirements in a typical SMT processor. As in CRTR, we have adopted a slightly different policy. When the distance between the two threads is below the threshold imposed by the slack, only the master thread is allowed to fetch new instructions. Contrarily, when the distance is above the threshold, the fetch priority is given to the slave. However, in order to use all the available bandwidth, if the slack is not satisfied but for some reasons the master thread cannot fetch more instructions, we allow the slave thread to fetch. In the remaining stages of the pipeline such as decode, issue, execution and commit, the used policy is FIFO.

We can also experience a noticeable performance degradation if the master thread fetches enough instructions to completely fill the shared ROB. This happens since the master thread runs some instructions ahead of the slave. In this scenario, the master thread cannot fetch more instructions because of the previously described fetch policy, neither does the slave because the ROB is full. So,

until any ROB entry is released, the two threads are stalled and cannot fetch new instructions.

We address this problem by keeping a percentage of free entries in the shared ROB for the slave. This way, we avoid both threads to stall due to ROB contention. Our experimental results show that 20% of total ROB's free entries is the best case in order to reduce this penalty.

An alternative approach would be to use a private ROB for each thread (or a static partitioning). However, the requirements of the master and slave threads are changing constantly due to the slack mechanism, branch mispredictions and long latency memory operations. In this scenario, a static partitioning is not able to maximize the use of all the available ROB entries. Therefore, a fully shared ROB is the best approach to the architecture presented in REPAS.

4.3.4 Reliability in the Forwarding Logic

In our design, the integrity of the information within structures as caches or additional buffers is protected by means of ECC codes. However, a traditional issue derived from the use of queues to bypass data are the potential problems arising from errors in the forwarding logic. An error in the LSQ forwarding logic in the master executing a load instruction might cause an incorrect bypass to the corresponding slave's load. If this happens, the slave thread would consume wrong values from the LVQ leading to a SDC (Silent Data Corruption).

To address this potential problem, in REPAS we use a double check: the slave thread compares the load values obtained by means of its own LSQ with the corresponding values in the LVQ. This way, if either the forwarding logic of the master or the slave fail, this check will detect a mismatch in the values signaling a fault. This mechanism results appropriate to assure the correction of the data forwarding in the LSQ. Nevertheless, there are some environments in which the coverage could not be considered good enough. In those cases, another mechanism at micro-architecture level as proposed in [13] could be applied, achieving almost a 100% AVF reduction while affecting performance in just 0.3%.

4.4 Evaluation Results & Analysis

4.4.1 Simulation Environment

The methodology used in the evaluation of this work is based on full system simulation. We have implemented all the previously described proposals by extending the multiprocessor simulator GEMS [47] from the University of Wisconsin-Madison.

TABLE 4.1: Characteristics of the evaluated architecture and used benchmarks.

(a) System characteristics

16-Way Tiled CMP System		**Cache Parameters**	
Processor Speed	2 GHz	Cache line size	64 bytes
Execution Mode	Out-of-order	**L1 cache**	
Max. Fetch / retire rate	4 instructions / cycle	Size	64KB
ROB	128 entries	Associativity	4 ways
FUs	6 IALU, 2 IMul 4 FPAdd, 2 FPMul	Hit time	1 cycle
Consistency model	Total Store Order (TSO)	**Shared L2 cache**	
Memory parameters		Size	512KB/tile
Coherence protocol	Directory-based MOESI	Associativity	4 ways
Write Buffer	64 entries	Hit time	15 cycles
Memory access time	300 cycles	**Fault tolerance parameters**	
Network parameters		LVQ	64 entries
Topology	2D mesh	SVQ	64 entries
Link latency (one hop)	4 cycles	RVQ	80 entries
Flit size	4 bytes	BOQ	64 entries
Link bandwidth	1 flit/cycle	Slack Fetch	256 instructions

(b) SPLASH-2 + Scientific Benchmarks

Benchmark	**Size**	**Benchmark**	**Size**
Barnes	8192 bodies, 4 time steps	Raytrace	10Mb, teapot.env scene
Cholesky	tk16.0	Tomcatv	256 points, 5 iterations
FFT	256K complex doubles	Unstructured	Mesh.2K, 5 time steps
Ocean	258 x 258 ocean	Water-NSQ	512 molecules, 4 time steps
Radix	1M keys, 1024 radix	Water-SP	512 molecules, 4 time steps

(c) ALPBench + Server Applications

Benchmark	**Size**	**Benchmark**	**Size**
FaceRec	ALPBench training input	Speechrec	ALPBench training input
MPGDec	525_tens_040.mv2	Apache	100,000 HTTP transactions
MPGEnc	Output from MPGDec	SpecJBB	8,000 transactions

Our study has been focused on a 16-core CMP in which each core is a dual-threaded SMT, which has its own private L1 cache, a portion of the shared L2 cache and a connection to the on-chip network. The architecture follows the Total Store Order (TSO). The coherence protocol is a directory-based MOESI. The main parameters of the architecture are shown in Table 4.1(a). Among them, it is worth mentioning the 2D-mesh topology used, as well as the 256-instruction slack fetch as a result of the sensitivity analysis performed in Section 4.4.2.

The sizes and parameters for the studied applications are reflected in Table 4.1(b) and Table 4.1(c), respectively. We have performed all the simulations with different random seeds for each benchmark to account for the variability of multi-threaded execution. This variability is represented by the error bars in the figures, enclosing the confidence interval of the results. Among the evaluated benchmarks we have applications from SPLASH-2 and ALPBench and other server applications. We refer to Chapter 3 for further details.

For comparison purposes we have implemented several previous proposals. As explained in Section 4.2.1, DCC incurs in an additional performance degradation when it is ported from a shared-bus to a direct-network. The use of shared-buses will be no longer possible in future CMP architectures due to area, scalability and power constraint issues. Therefore, we compare our proposed REPAS with DCC when a direct network such as a 2D-mesh is used. Additionally, we compare REPAS with the performance of SMT-dual and DUAL. SMT-dual models a coarse-grained redundancy approach which represents a 16-core 2-way SMT architecture executing two copies (A and A') of each studied application. Within each core, one thread of A and one thread of A' are executed. As mentioned in [70], this helps to illustrate the performance degradation occurred within a SMT processor when two copies of the same thread are running within the same core. DUAL represents a 16-core non-SMT architecture executing two copies of the same program. In the case of 16-threaded applications it means that each processor executes 2 threads (1 thread of every application). In DUAL, the OS is responsible for the schedule of the different software threads among the different cores.

4.4.2 Slack Size Analysis

Before moving on to the performance study of REPAS, firstly, this subsection analyses the size of the slack parameter in REPAS.

As explained before, the slack fetch mechanism maintains a constant delay between master and slave threads. This delay results in a performance improvement (due to thread-pairs cooperation) because of factors such as the reduction of the stall time for L1 cache misses in the slave and the better accuracy in the execution of slave's branches thanks to the BOQ. From this perspective, we would choose to use a slack as big as possible.

However, a larger size of the slack also requires an increase in the size of structures like the SVQ or the LVQ to avoid stalls. Furthermore, in a shared-memory environment, a large slack causes the average life latency of a store (the time spent between the execution of the store and its validation) to be increased.

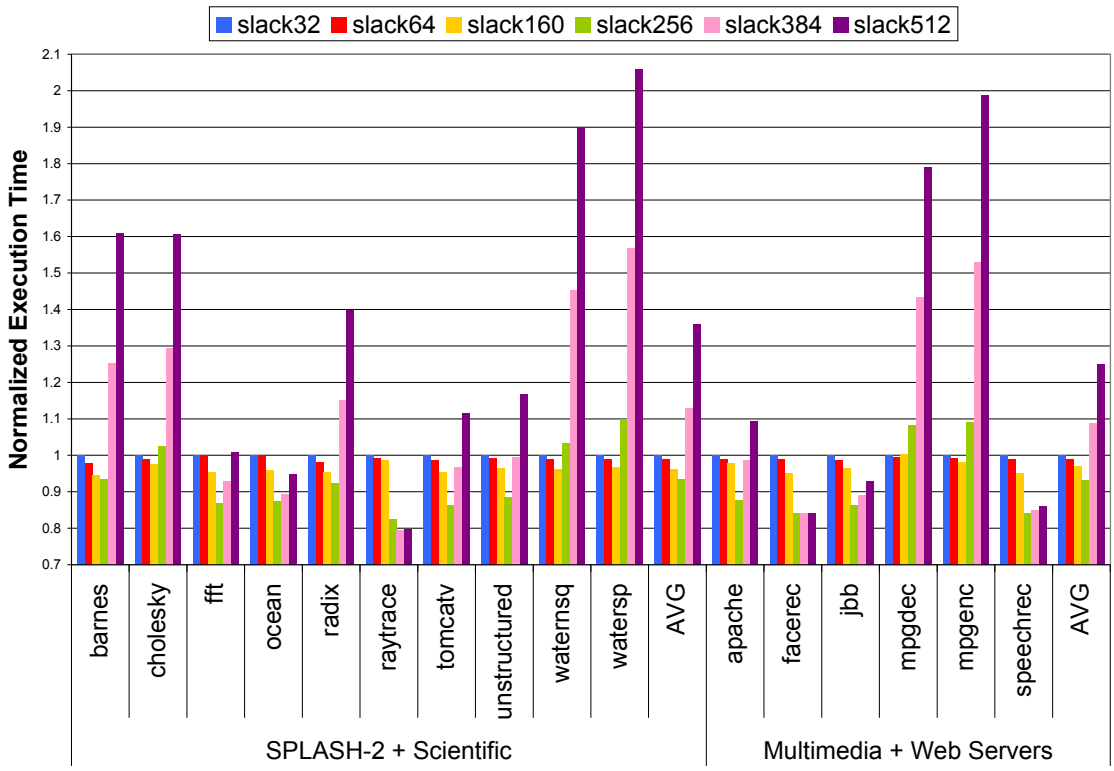


FIGURE 4.7: Sensitivity analysis for the optimal size of the slack.

This negatively affects performance because unverified blocks cannot be shared or replaced from cache. Figure 4.7 shows a sensitivity analysis for different sizes of the slack. The slack is measured in a number of fetched instructions between the master and the slave thread. The bars are normalized with respect to a 32 slack size. As it is shown, the increase of the slack size to 256 instructions results in a noticeable performance improvement. However, further increasing of the slack size, results counterproductive. On average, a slack of 256 instructions is 7% better than a slack of 32. Therefore, for subsequent experiments we will use 256 as our target slack.

4.4.3 Execution Time Overhead of the Fault-Free Case

We compare our proposed REPAS architecture with CRTR by using the alternative mechanisms, atomic synchronization and atomic speculation, as explained in Section 4.2.2. As many other previous proposals [37, 55, 83], we initially present the performance results of our mechanism in a fault-free environment in order to quantify the execution time overhead for the common case.

Figure 4.8 plots the results of REPAS normalized with respect to a 16-core system in which there is not a fault tolerant mechanism. CRTR_sync refers to the atomic synchronization mechanism for CRTR, and CRTR_spec does to the atomic speculation mechanism. As derived from Figure 4.8, REPAS outperforms CRTR_sync for both groups of benchmarks (scientific and multimedia/web) by 13% and 6% respectively, whereas the execution time overhead rises to 25%, on average, for all the studied benchmarks.

The main source of degradation in CRTR_sync comes from the frequent synchronizations between master and slave threads as a result of the execution of atomic instructions and memory fences. This effect can be better observed in those benchmarks with more synchronizations such as Ocean, Raytrace and Unstructured, in which the performance exhibited by CRTR_sync is even worst.

As it was expected, CRTR_spec outperforms CRTR_sync because of the effectiveness of the speculative mechanism. However, in benchmarks with highly

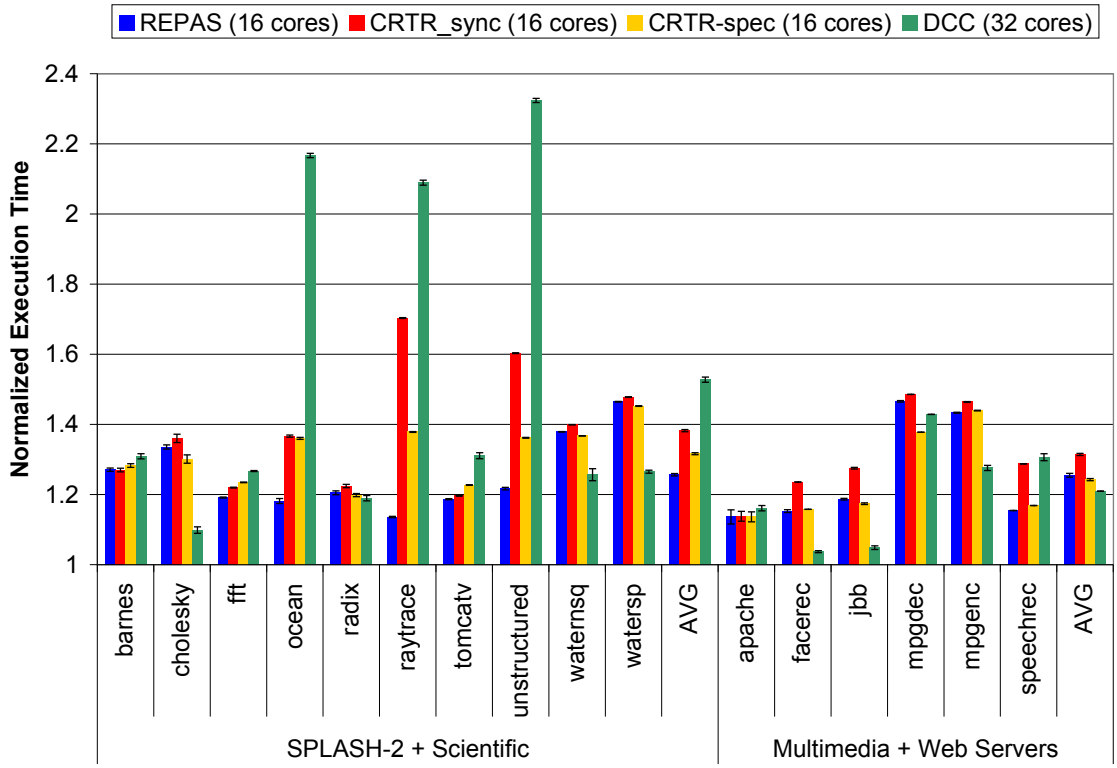


FIGURE 4.8: Execution time overhead over a non fault-tolerant 16-core architecture.

contended locks such as Ocean, Raytrace and Unstructured, the number of rollbacks due to miss-speculation has a significant impact on performance in relation to REPAS. On average, REPAS is 6% faster than CRT_spec for SPLASH-2 benchmarks, although for Multimedia and Web Server applications CRTR_spec shows a performance similar to REPAS, benefited from the low synchronization exhibited by these applications.

The performance degradation reported for DCC when evaluated within a shared-bus is roughly a 5% for several parallel applications [37]. However, as explained in Section 4.2.1, this overhead is increased when a direct-network is used. As explained, the major source of degradation is related to the mechanism to assure the master-slave consistency which allows to avoid input incoherences.

As we can see in Figure 4.8, REPAS is able to outperform DCC by 27% for scientific applications. However, for multimedia and web servers benchmarks,

the performance exhibited by DCC is better than the performance of REPAS by 4%. The reason for this behaviour is explained by the poor performance of these applications in SMT architectures with respect to CMP architectures, a result which results consistent with the one by Sasanka *et al.* [80].

In any case, we have to remind that whereas REPAS uses SMT cores to provide fault-tolerance, DCC uses twice the number of cores as REPAS (in this evaluation, DCC uses 32 cores whereas REPAS uses 16 cores). This reduces the overall throughput of a system implementing DCC in more than 100% over a non fault-tolerant base case.

Finally, as we can see in Table 4.2, REPAS is 20% faster than SMT-dual on average which, at the same time, is slower than CRTR_sync and CRTR_spec by 10% and 17%, respectively. The performance degradation of SMT-dual is such because of the inadequate interaction of different threads in the same core. Nevertheless, in REPAS and CRTR threads collaborate (LVQ, SVQ, BOQ) in SMT-dual threads compete against one another for the resources of the core affecting performance. In the same way, DUAL affects performance noticeably. This is because in DUAL, threads must be re-scheduled by the OS to be executed in each core (note that we have 16 cores but 32 threads, 16 threads for every application). This adds an extra overhead of almost 2X in the computation. As a final remark, we can conclude that SMT approaches could benefit from a better performance than non-SMT approaches.

TABLE 4.2: Average normalized execution time for the studied benchmarks.

	REPAS (16 cores)	CRTR_sync (16 cores)	CRTR_spec (16 cores)	DCC (32 cores)	SMT-dual (16 cores)	DUAL (16 cores)
Normalized Exec. Time	1.25	1.35	1.28	1.40	1.45	1.88

4.4.4 Performance in a Faulty Environment

We have shown that REPAS introduces an overhead in a fault-free scenario despite of outperforming several previous proposals. Nonetheless, REPAS guarantees the

correct execution of shared memory applications even in the presence of soft errors. The failures and the necessary recovery mechanisms introduce an additional overhead that we study now.

Figure 4.9 shows the execution time overhead of REPAS under different fault rates normalized with respect to a non-faulty environment case. Failure rates are expressed in terms of faulty instructions per million of cycles per core. For a realistic fault ratio, the performance of REPAS is barely affected so, for this experiment, we have used fault rates which are much higher than expected in a real scenario in order to show the kindness of the proposed architecture[3].

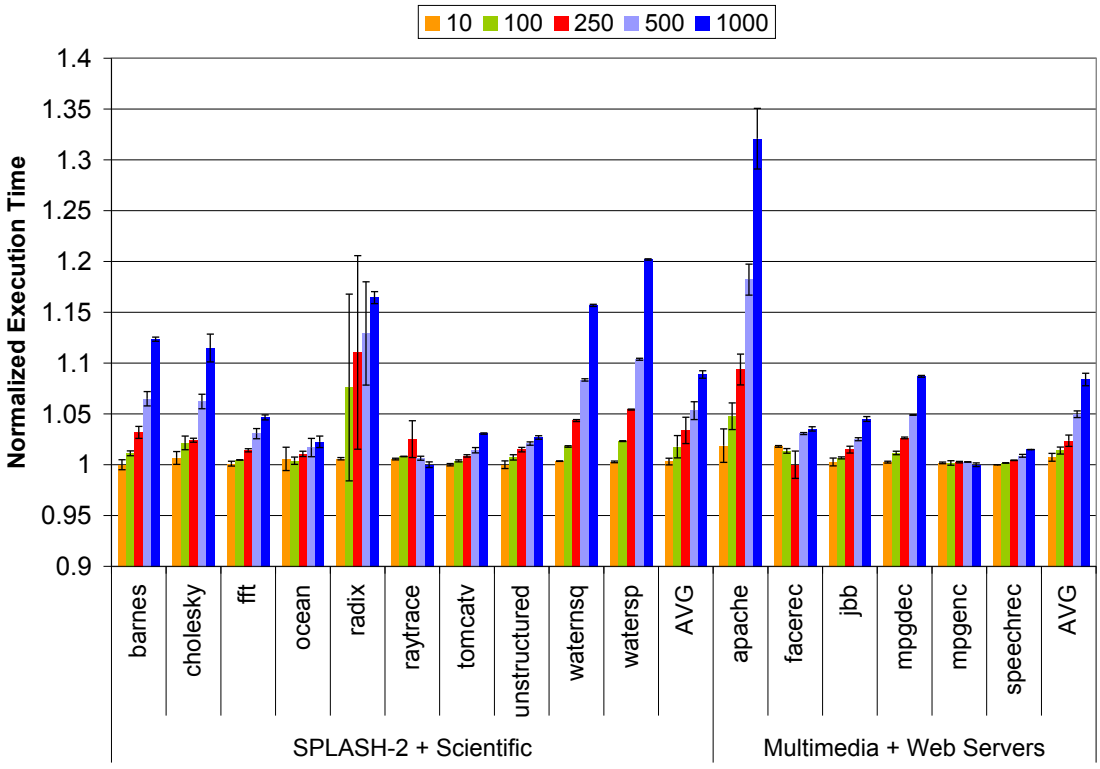


FIGURE 4.9: REPAS overhead under different fault rates (in terms of faulty instructions per million per core).

As we can see, REPAS is able to tolerate rates of 100 faulty instructions per million cycles per core with an average performance degradation of 1.6% in the execution time in comparison to REPAS in a non-faulty environment. Only when

[3]E.g., a ratio of 10 failures per million cycles per core is equivalent to a MTTF of $3,125 * 10^{-6}$ sec. for the proposed architecture.

the fault ratio is increased to the huge (and unrealistic) amount of 1000 failures per million cycles, performance shows a noticeable degradation of 8.6%. As expected, performance degradation rises almost linearly with the increase of the fault ratio, although it still allows the correct execution of all the studied benchmarks.

The time spent on every recovery varies across the executed benchmark. This time includes the invalidation of all the unverified blocks and the rollback (bypass the safe state of the slave thread to the master) of the architecture up to the point where the fault was detected. On average this time is 80 cycles. In contrast, other proposals such as DCC spend thousands of cycles to achieve the same goal (10,000 cycles in a worst-case scenario). This clearly shows the greater scalability of REPAS in a faulty environment.

4.4.5 Sharing Unverified Blocks

As initially implemented, REPAS does not allow the sharing of unverified blocks. This conservative constraint avoids the propagation of errors among cores. However, it is not expected that it imposes a high performance degradation, since the verification of blocks is quite fast (in the order of hundred cycles). On the contrary, DCC [37] is based on a speculative sharing policy. Given that blocks are only verified at checkpointing creation intervals (i.e., 10,000 cycles), avoiding speculative sharing in DCC would degrade performance in an unacceptable way.

For comparison purposes, we have studied the effect of sharing unverified blocks in REPAS. The mechanism is straightforward to implement: accept forward requests for blocks in unverified state. However, since we do not support checkpointing capabilities as DCC, to avoid unrecoverable situations, cores obtaining speculative data cannot commit. This way, if a fault is detected by the producer of the block, all the consumer cores can recover by flushing their pipeline in a similar way as it is done when a branch is mispredicted. An additional disadvantage is that the producer of the block must send a message indicating whether the shared block is faulty or not, increasing the network traffic. Luckily, the sharing information is gathered from the sharers list as in a conventional MOESI protocol, so we do not need additional hardware to keep track of speculative sharings.

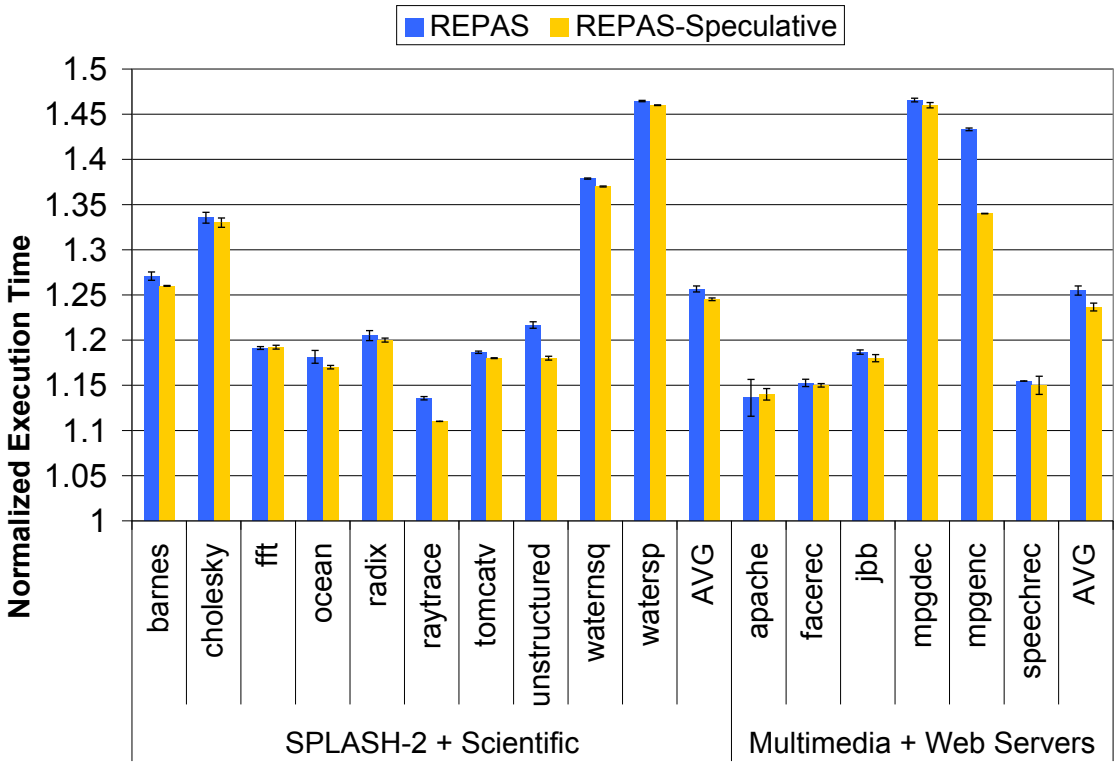


FIGURE 4.10: Normalized execution time with and without the speculative mechanism.

Finally, we have not considered to migrate unverified data speculatively, since an expensive mechanism would be necessary to keep track of the changes in the ownership, the sharing chains as well as the original value of the data block (for recovery purposes).

As we can see in Figure 4.10, the performance improvement for the speculative mechanism is negligible (around 1% on average). Only benchmarks such as Ocean, Raytrace, Unstructured and MPGEnc, speculation obtains a noticeable improvement (up to 10% for MPGEnc). Table 4.3 reflects that speculations are highly uncommon. Furthermore, if we consider the time to verification of speculative blocks it can be seen that, on average, we could benefit from around 100 cycles, although they cannot be fully amortized because pipeline is closed at commit. This explains why speculative sharings do not obtain many benefits in REPAS. Overall, the speculative sharing mechanism seems inadequate for the studied benchmarks,

TABLE 4.3: Number of speculative sharings and time needed to verify those blocks.

BENCHMARK	Speculations	Time to Verification
Barnes	12860	92.5
Cholesky	5758	161.5
FFT	128	94.5
Ocean	13786	94.5
Radix	710	82
Raytrace	37031	92
Tomcatv	250	91
Unstructured	223524	107
Water-NSQ	1585	98
Water-SP	339	89.5
Apache	135	99.5
Facerec	0	-
JBB	877	94.5
MPGDec	0	-
MPGEnc	48997	123.5
Speechrec	0	-
AVG	**-**	**101.875**

since it is not worth the incremented complexity in the recovery mechanism of the architecture.

4.4.6 L1 Cache Size Stress

An unverified block cannot be evicted from L1 cache since potentially faulty blocks would go out of the SoR (Sphere of Replication). In an environment with high pressure over the L1 cache, this can cause a performance degradation due to the unavailability of replacements to be completed. In this Section, we study how REPAS behaves with different configurations.

It could be expected that the stress of cache size would impact negatively on the performance of REPAS. However, the results show that this forecast is not fulfilled. Figure 4.11 represents the execution time of REPAS for different L1 cache configurations. Each set of bars is normalized with respect to the case base with the same configuration.

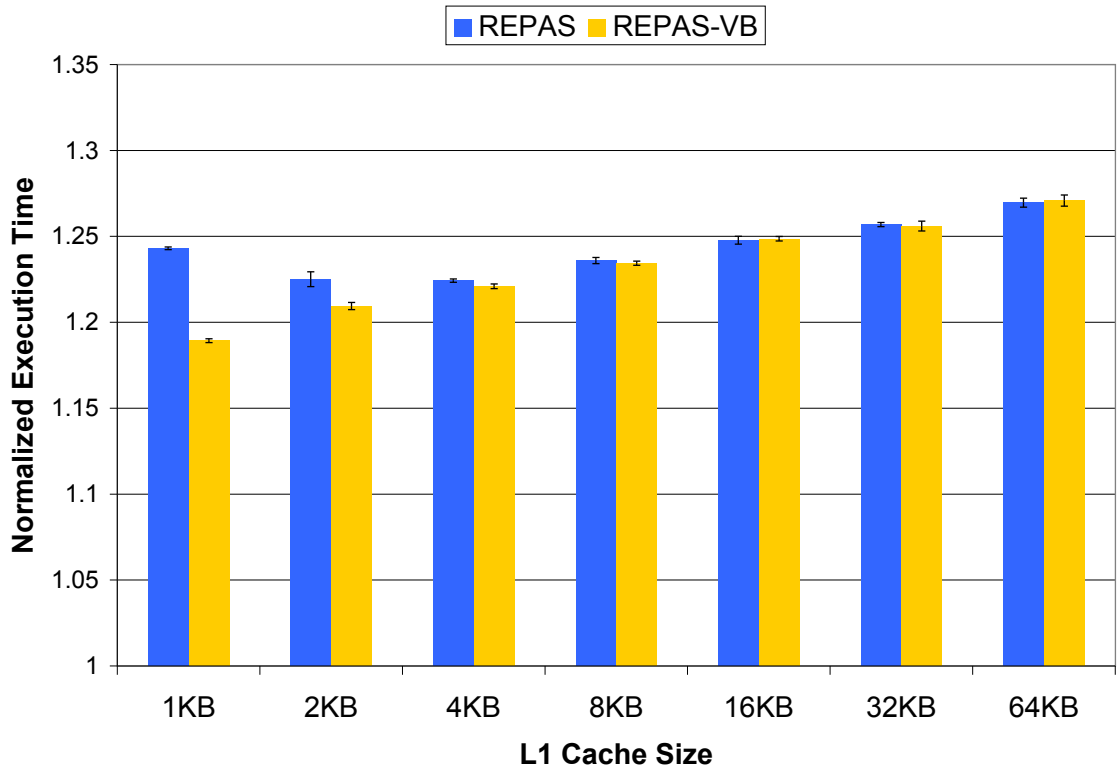


FIGURE 4.11: Normalized execution time for different L1 cache sizes with and without a Victim Buffer.

Contrarily to expected, smaller caches do not degrade performance in REPAS but even improve it in comparison with the base case (1KB, 2KB and 4KB perform better than the 64KB configuration in comparison with the base case with the same configuration). The reason for this behaviour is subtle but it can be easily explained if we consider to the REPAS mechanism. As we said before, a smaller cache penalizes REPAS because of the increased latency of the L1 replacements. However, a smaller cache also penalizes the architecture due to the increased L1 cache miss ratio. The key point here is that, whereas the processor is stalled on a cache miss in the base case, in REPAS, L1 misses (or master stalls in general) are used by the slave thread to continue executing program instructions, thus making forward progress.

Finally, an approach to allow the eviction of unverified blocks from L1 to L2 is to use a small Victim Buffer. With this mechanism, L1 cache replacements of these blocks are performed out of the critical path. As we can see in Figure

4.11, the Victim Buffer improves the performance for 1KB, 2KB and 4KB. For the rest of configurations there are not any noticeable gains because the number of unverified blocks to replace from L1 cache is very low. Experimental results show that the maximum size needed for the VB is 14 entries which we consider acceptable without devoting too much hardware.

4.5 Concluding Remarks

Processors are becoming more susceptible to transient faults due to several factors such as technology scaling, voltage reduction, temperature fluctuations, process variation or signal cross-talking. Although there are many approaches exploring reliability for single-threaded applications, shared-memory environments have not been thoroughly studied.

Proposals like DCC or Reunion use DMR (Dual Modular Redundancy) to provide fault tolerance in microarchitectures. However, they impose a 2X hardware overhead, an unacceptable result for manufacturers which claim for a 10% maximum extra area impact. Hence, in this Chapter we propose REPAS: *Reliable Execution for Parallel ApplicationS in tiled-CMPs*, a novel RMT approach to provide transient fault detection and recovery in parallel and shared-memory applications.

Whereas other proposals use large amounts of extra hardware, other RMT architectures perform reliable computation by redundant thread execution (master and slave) in SMT cores. Therefore, the hardware overhead is kept low. However, the architectural support for shared-memory applications has remained underexplored so far. In this work, we show that atomic operations induce a serialization point between master and slave threads, a problem which may be minimized by means of speculation in the consistency model. Although this solution requires both a change in the way atomicity is implemented and a hardware increase to support the speculation, the degradation in low-to-medium contention benchmarks remains moderate. However, in scenarios with high contention the performance

is severely affected. In REPAS we effectively avoid this overhead due to synchronization or miss-speculations by eager updates of the L1 cache.

We have implemented our solution in a full-system simulator and presented the results compared to a system in which no fault-tolerant mechanisms have been introduced. We show that, in a fault-free scenario, REPAS reduces the execution time overhead down to 25%, outperforming CRTR, a traditional RMT implementation. We have also compared REPAS with DCC, showing some gains in certain applications but looses in others. Nonetheless, REPAS uses half the number of cores than DCC, providing a better throughput. We have also evaluated the performance of REPAS in a faulty environment, showing an increase of just 2% of execution time with a huge fault ratio of 100 faults per million of cycles per core. This ratio is much higher than expected in a real scenario, so negligible slowdown is reported in a realistic faulty environment.

Finally, we have performed a L1 cache size stress in order to study the behaviour of REPAS due to its inability to evict blocks from cache until verification. Results show that even with smaller cache sizes, the performance degradation of REPAS is kept in acceptable margins. Additionally, a Victim Buffer to hold unverified blocks has been used in REPAS showing slight performance improvement (up to 4%) for configurations which highly stress the L1 cache.

Chapter 5

LBRA: A Log-based Redundant Architecture

SUMMARY:

CMOS scaling exacerbates hardware errors making reliability a big concern for recent and future microarchitecture designs. Mechanisms to provide fault tolerance in architectures must accomplish several objectives such as low performance degradation, power dissipation and area overhead. Several studies have been already proposed to provide fault tolerance for parallel codes. However, these proposals are usually implemented over non-realistic environments including the use of shared-buses among processors or modifying highly optimized hardware designs such as caches.

Our attempt to address this multiple challenge is an architectural design called LBRA (Log-Based Redundant Architecture). Based on a Hardware Transactional Memory architecture, LBRA executes redundant threads which communicate through a pair-shared virtual memory log located in cache. Our initial version of LBRA executes these redundant threads in SMT cores. To avoid the performance penalty inherent to this architecture, we propose to decouple their execution in different cores, solving the inter-core communication by means of a log buffer empowered by a simple prefetch strategy. Simulation results using a variety of scientific and multimedia applications show that the execution time overhead of our best design is less than 7% over a base case without fault tolerance. Additionally, we show that LBRA

outperforms previous proposals that we have implemented and evaluated in the same framework.

5.1 Introduction

Being reliability a major concern for hardware architects, several mechanisms to detect and recover from faults have already been implemented in microarchitectures. This is the case of ECC, which is nowadays applied in large CAM arrays such as caches or RAM memories. Unfortunately, ECC cannot be extensively used through out all hardware structures. On the contrary, architectural-level mechanisms provide a more flexible framework in which multiple hardware structures are covered in comparison to cycle-level techniques which are focused on single units.

As we showed in Chapter 2, one of the most straightforward and studied mechanisms to provide fault tolerance is the use of physical and/or time redundancy, in which programs are executed multiple times and outputs are compared in order to detect errors. Among these works we can distinguish two different trends: (1) those in which memory is not updated until the values have been satisfactory checked like the CRT(R) family [25, 55, 70, 107]; and (2) those in which, once a fault is detected, the state of the architecture in rolled-back to a previous known-to-be-safe checkpoint like DCC [37] or HDTLR [69]. In the first case, performance is affected given the fact that forward progress may be stalled until verification is accomplished. In the second case, the major drawback is the synchronization between redundant executions, which includes stopping execution, sharing and comparing architectural states and, finally, resuming execution.

In order to be feasible, a fault tolerant architecture should degrade performance as less as possible and should also require an area overhead not larger than 10% [74]. With these goals as motivation, in this Chapter we explore the use of already existing log-based hardware transactional memory (HTM) systems as a

novel way to provide fault tolerance. Thus, we present LBRA: Log-Based Redundant Architecture for reliable parallel computation. What we propose is to build a fault tolerant architecture by using a current state-of-the-art HTM system with slight modifications, something which, to the best of our knowledge, has not been previously studied. The main idea is to execute redundant copies of the same software thread in two different hardware contexts which are executed within the same SMT core. In this work we have chosen LogTM-SE [115], an elegant HTM design which performs both *eager version management*, by updating memory in place and keeping the old values in a virtual memory space called log, and *eager conflict detection*.

The proposed LBRA provides high flexibility, allowing the programmer to manually declare areas of the program to be protected or not. Program instructions in these areas are divided into virtual execution groups called *pseudo-transactions* (p-XACTs) or chunks [15]. The master thread executes p-XACTs as regular instructions but, additionally, it keeps the results of its progress in a pair-shared log. By means of this log, the slave verifies that the results produced by the master are correct. We provide a highly decoupled environment since the master is allowed to execute multiple p-XACTs without verification, something which is carried out off the critical path by the slave thread. This high decoupling allows the latencies to be hidden, due to memory or inter-core communication.

The major contributions of the LBRA proposal are:

- An architecture design which, on top of a Hardware Transactional Memory system and SMT cores, provides fault tolerance in a parallel point-to-point network environment.
- A study of the implications of running redundant threads in different cores. Thus, we avoid performance degradation due to resource contention associated to SMT-based proposals running pairs of redundant threads.
- A set of hardware mechanisms to reduce and/or hide the inter-core communication latency. These mechanisms include a log buffer combined with a simple prefetch strategy and slight modifications of coherence actions.

- A detailed comparison among the already proposed architecture design and state-of-the-art proposals within the same framework. For this evaluation, we make use of a great variety of parallel benchmarks executed in both SMT and non-SMT cores.

The remainder of this Chapter is organized as follows: Section 5.2 briefly introduces Hardware Transactional Memory and explains how it can be adapted for fault tolerance purposes. In Section 5.3 we discuss the implementation details of LBRA in 2-way SMT cores, whereas in Section 5.4 we extend it to redundant regular cores as a way to reduce the performance penalty inherent to the use of simultaneous multithreading. The evaluation setup and analysis are described in Section 5.5. Finally, Section 5.6, summarizes the main conclusions of this work.

5.2 HTM Support for Reliable Computation

Our proposed LBRA approach is built upon the top of a LogTM-SE [115] system, a hardware implementation of Transactional Memory. This Section describes how we may provide fault tolerance to the system by adding several modifications to its behaviour with a modest hardware overhead.

5.2.1 Version Management

LogTM-SE offers an eager version management. This means that the values produced by transactions are directly updated in cache, where they become visible to the rest of the system. A lazy version management, on the contrary, does not expose updated values until commit. Generally speaking, an eager mechanism performs better than a lazy one in case rollbacks are infrequent. Thus, since we would only apply a rollback in case of a detected fault, it seems that the use of an eager version management is the most appropriate one for our purposes.

5.2.1.1 Input Replication

LBRA could be classified as a Redundant Multi-Threading (RMT) approach. In RMT systems, two hardware threads (commonly called master and slave), redundantly execute the program instructions to provide fault tolerance within SMT or independent processor cores. Note that, unlike true software threads, each redundant thread pair appears to the operating system as a single one.

As we explained in Chapter 2, in RMT systems one of the most important issues is *input replication* which defines how redundant threads or executions observe the same data. Since master and slave thread execution is not lockstepped [5], the execution of redundant memory instructions would probably lead to input incoherences. In order to solve this problem, in our proposal we extend the functionality of the log (a memory space allocated in virtual memory already implemented in LogTM-SE) as follows: for each load instruction, the master thread keeps the result of its execution in the log. This way, slave load instructions are served through the log where they obtain the same values as its master-pair, avoiding thus input incoherences. Note that, as the log is written at instruction commit, it will only keep instructions of the correct execution path and in program order.

5.2.1.2 Output Comparison

The *output comparison* defines how the correctness of the computation is assured in RMT systems. In our LBRA approach, we define the output comparison granularity at a pseudo-transaction (p-XACT) level. A p-XACT defines the unit of work which is considered to be either incorrect or correct, depending on whether faults have been detected within its execution or not.

The semantic and execution of a p-XACT is quite different from a regular transaction (XACT) in LogTM-SE. Firstly, whereas traditional XACTs are manually coded in the application, p-XACTs are dynamically created in execution time and their length is variable, as we will see later. This provides a great flexibility, making redundancy easy to turn on and off. Secondly, a p-XACT does not ensure isolation and/or atomicity. This way, dirty memory blocks are shared as

in a non-transactional environment, relying on other synchronization mechanisms such as locks or barriers to assure correction.

The execution of a program in LBRA is as follows. First, the master starts the execution of a new p-XACT. This implies the allocation of a new section in the log and the initialization of the registers which hold R/W signatures. These signatures summarize the read and write sets, and are used to determine, for a given address, if a block was previously accessed or not. Eventually, a mechanism would trigger a signal indicating the end of the current p-XACT. We define this event as the *commit* of the p-XACT. The commit is completely local and it does not require communication outside the actual core, what enables this mechanism to be remarkably fast. However, unlike in the original LogTM-SE implementation, this mechanism does not clear the R/W signatures or resets the log pointer (this will be carried out by the slave thread). Finally, the active transaction is considered as finished and the following program instructions are executed within a new p-XACT.

The task of the slave is to assure the correct execution of all the work done by the master. To accomplish this, the slave thread redundantly executes the p-XACTs committed by the master obtaining memory values from the log. At the end of the p-XACT, the slave performs what we called the *consolidation*. In the consolidation process, the architectural state of master and slave threads are compared to assure that the produced values are correct. For this purpose, we follow a similar approach as in [37], where signatures summarizing the computational work are compared. The master thread creates an in-flight signature which is saved in the *Verification Signature* at commit for every p-XACT (see Figure 5.1 for additional implementation details). Then, in the consolidation, the slave compares its own signature with the *Verification Signature*. Upon a match, the execution of the p-XACT is correct. Therefore, the signature registers and the log pointer can be cleaned out. Finally, the slave performs a backup of its register file which is now considered correct. On the contrary, if the consolidation process results in a mismatch, the recovery mechanism must be triggered. The last backup of the registers would be used to restore the architectural state of the machine. Note, though, that faults can be detected before consolidation. This happens if

the slave detects a mismatch in the addresses accessed in the log by load or store instructions, as we will further describe in Section 5.3.1.3.

5.2.2 Dependence Tracking

p-XACTs rely on software mechanisms to ensure atomicity and isolation. As blocks are allowed to be shared, potential faults could be spread across the system, so we need to keep track of these interactions. Although conflict detection is not engaged in p-XACTs, we find this mechanism already implemented in LogTM-SE very suitable to keep track of potential faulty shared blocks.

LogTM-SE provides eager conflict detection by means of the coherence protocol, decoupling the mechanism from caches by using R/W signatures. External requests arriving to a core are checked through these signatures and, on a possible conflict[1], requests are NACKed. What we propose is to use these signatures to maintain a pair of per-transaction registers called *Producer Register* and *Consumer Register*, see Figure 5.1. The Producer and Consumer registers keep the transaction identifiers involved in the data sharing of all the cores in the system.

The proposed mechanism works as follows. A core receiving a forward request checks its write signatures from all active p-XACTs (those which have been already committed by the master or are still in execution). For a positive match in an active p-XACT, the core updates the *Producer Register* storing the transaction *id* for the involved core. In the same way, the requestor of the block, when obtaining a response, updates its *Consumer Register* indicating the core and transaction *id* produced by the previously obtained block. All the required information is obtained from memory request messages.

The functionality of these registers is twofold. First, when a fault is detected the *Producer Register* is used in the recovery process to abort all the p-XACTs involved since their states are potentially corrupted, as we will see later. Secondly, the *Consumer Register* is used to provide an order in the consolidation mechanism,

[1] A conflict occurs when an address appears in the write-set of two transactions or the write-set of one and the read-set of another [115].

needed to avoid SDCs (Silent Data Corruptions). We will address this issue in Section 5.3.3.

5.3 LBRA Implementation Details

One of the major drawbacks in previous RMT approaches is the synchronization between redundant threads. As a measure to amortize the latency of comparing redundant executions, long checkpoint intervals are needed. Our goal is to eliminate these latencies, independently of whether synchronizations are common or not. To achieve this, we use a decoupled approach by means of the capabilities of LogTM-SE and the ability to execute multiple p-XACTs before verification. The hardware additions needed to accomplish these goals are depicted in Figure 5.1.

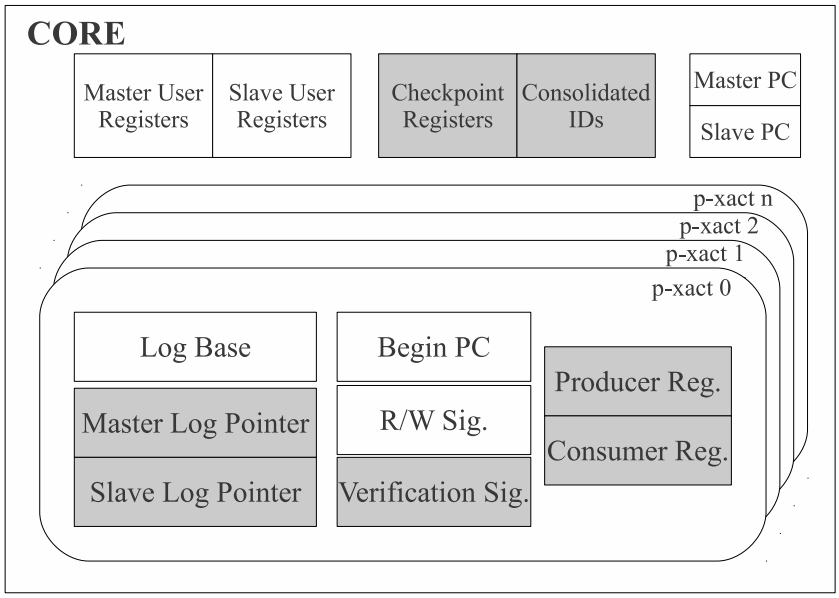


FIGURE 5.1: LBRA hardware overview. Shadowed boxes represent the added structures.

5.3.1 Accessing the Log

To provide access to the same log, both master and slave threads should share the memory space. To this end, unlike in true SMT threads, master and slave

threads appear to the OS as a single one as in [70]. The access to the log by both redundant threads is as follows.

5.3.1.1 Master Access

The master thread writes in the log through the *Master Log Pointer* as in a traditional LogTM-SE system. Note that this pointer is local to every p-XACT. At every memory operation, the master generates a new store instruction whose destination address is indicated by this pointer. This new store allows the system to satisfy input replication and output comparison as explained before in Section 5.2.1. As memory operations are logged in commit, the content of the log is structured in program order.

5.3.1.2 Slave Access

The slave accesses to the log are more complex and require a special treatment. In order to ensure input replication, each load access must be redirected to the log. For that purpose, at memory access time, the destination address of loads are switched with the *Log Slave Pointer* which indicates the location of the memory value previously read by the master thread. Then, the memory access is performed as usual and the log pointer is set to the next entry in the log.

In the case of stores, the mechanism differs slightly. Since slaves do not update memory, their stores become reads to the log. For this reason, the destination address is switched with the Log Slave Pointer address and the data value is retrieved from the log.

5.3.1.3 Log Content & Fault Detection Granularity

The size of the log is a major concern in our approach since its growth affects the available cache space for the application and, therefore, its performance. In this Section, we discuss how to decrease the size of the log by reducing the amount of

data to store, something which also affects the detection granularity. The different alternatives can be seen in Table 5.1.

TABLE 5.1: Alternatives in log content for loads and stores.

	Address	**Value**	**Old-value**	**Provides**
Loads	Yes	Yes	-	Input replication Fault detection in address calculation
	No	Yes	-	Input replication
Stores	Yes	Yes	Yes	Fault detection in address calculation Fault detection in value calculation Fault recovery
	Yes	No	Yes	Fault detection in address calculation Fault recovery

Faults in load address

To satisfy input replication, it is mandatory to include in the log every data value read by the master thread. However, the address of the load is optional. If we include it, we could detect faults affecting address calculation. But this presents two major drawbacks. First, the log size increases. And second, we increase the hardware pressure by adding an additional master-slave check on every load. Since our first goal is to reduce performance penalty, we choose to store the minimum information possible, i.e. only data values, and rely on the consolidation process to determine the correct execution of all the p-XACT.

Faults in store address

Likewise, we try to reduce the information we keep from stores to decrease the log size as much as possible. In order to recover from a fault, we rely on the LogTM-SE handler which restores modified memory values. For this purpose, we need to keep the address and the old value for every memory update in the log. Additionally, we could keep the current value to be stored. If so, a fault in the calculation of this value could be detected when the slave thread accesses the log. However, for the same reasons as for loads, we avoid to store the new value, waiting for faults to be detected at the consolidation phase.

5.3.2 Circular Log

LBRA provides a high decoupled execution of redundant threads. As a result, the forward progress of the master is rarely interrupted since the latencies inherent to the verification process are virtually hidden. For this purpose, the master thread is allowed to execute and commit several p-XACTs without verification. Meanwhile, the slave thread checks the correct execution of already committed p-XACTs and, as a final step, performs consolidations.

In order to allow multiple p-XACTs to be committed without verification, each one needs its own architectural support. This support includes R/W registers, verification signatures and log pointers as depicted in Figure 5.1. The amount of extra hardware is determined by the maximum number of in-flight p-XACTs allowed.

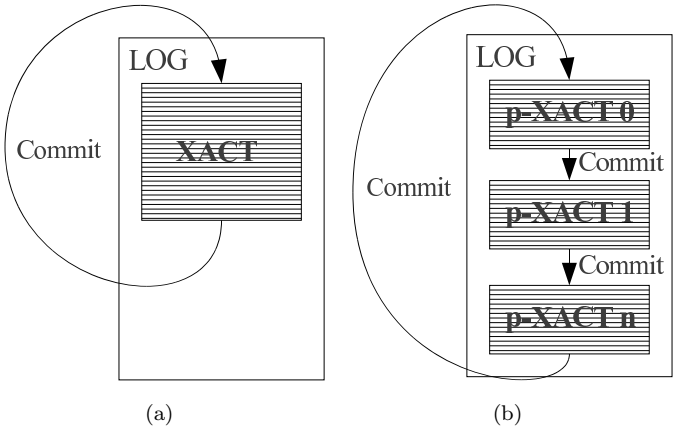


FIGURE 5.2: LogTM-SE and LBRA log management.

But the major implication derived from this support affects the management of the log. LogTM-SE only allows one transaction to be executed at a time [2]. After commit, signatures are cleared and log pointers are reset. Thus, the next XACT will start writing the log from the beginning of the reserved memory space, as we can see in Figure 5.2(a). However, in LBRA the log must be preserved for the slave thread. Therefore, instead of resetting log pointers, after the commit of a p-XACT, the following one starts writing from the last entry used by the previous

[2]It also allows several XACTs to be merged, becoming a logical one.

p-XACT, as we can see in Figure 5.2(b). Only when the pointer reaches the limit of the memory reserved for the log, it is set to its beginning. In a nutshell, in LBRA the log grows circularly through all the virtual space reserved for it.

5.3.3 In-order Consolidation

In our approach, memory blocks are updated in place (L1 cache) and allowed to be shared even before consolidation takes place. This eager approach allows fast commits and adequate performance, since faults can be considered as the uncommon case. However, this mechanism affects the consolidation order of p-XACTs since, if additional mechanisms are not implemented, faults could be spread all around the system.

It is clear that if a p-XACT p_i has consumed data produced from another p-XACT p_j, the consolidation of p_i cannot take place before the consolidation of p_j. Otherwise, a faulty block produced in p_j would be silently consolidated in p_i. To keep track of these dependencies, we introduce the Consumer and Consolidated-Ids registers, as explained in Section 5.2.2, which gather the information provided by the coherence protocol. To achieve this, memory coherence messages in our approach are extended to include the p-XACT identifier providing the data, which are used by the requestor to fill the Consumer register, and the last consolidated p-XACT identifier.

The in-order consolidation process works as follows. After completing the verification of state, the slave thread checks the Consumer vector for the current p-XACT. If it is empty, it means that this p-XACT has not consumed data from any other p-XACT, so the consolidation process may take place without any additional checks. If the Consumer register is not empty, then, for every dependence, the slave checks if the producer p-XACT has already been consolidated by checking the Consolidated register. If all the dependencies satisfy this condition, then the p-XACT is finally consolidated. If not, we initiate a lookup mechanism. The slave thread requests its producers to supply the last consolidated *id* until all the dependencies are satisfied.

5.3.3.1 Cycle Avoidance

There exists a danger of deadlock in the consolidation process if we allow cycles to be formed. For example, let us consider the case in which p_i is the producer of p_j which, at the same time, is the producer of p_k and, finally, p_i consumes data from p_k. In this case, none of the three p-XACT could be consolidated since a cycle has been created. Although this case is rare, we need to present a mechanism to avoid it.

Our goal is to create a DAG (Directed Acyclic Graph). DAGs assure that a topological order exists although this order, in general, is not unique. Therefore, we implement a simple policy: we disallow situations in which a p-XACT is both producer and consumer of other p-XACTs at the same time. When a master thread which is already a producer receives data produced by a p-XACT, the active p-XACT is forced to commit and a new one is started before consuming these data. Likewise, if a consumer p-XACT is requested to provide data (becoming a producer), it is forced to commit and the dependence is created in a new p-XACT. This guarantees that no cycles can be created avoiding consolidation deadlocks.

5.3.4 Fault Recovery in LBRA

Upon fault detection the recovery mechanism is triggered. In our approach, this mechanism is taken by a combination of both software and hardware processes for local and global recovery which act on the youngest p-XACT of the core. The correctness of the proposed mechanism is proved since dependencies form a DAG, so a topological order can be established.

5.3.4.1 Local Recovery

The local recovery is the rollback to a safe state previous to the execution of a faulty p-XACT in a core. For this process we rely on the software approach proposed in LogTM-SE to abort transactions. This software mechanism writes back the old values to their appropriate addresses from the log. After that, the transactional

hardware of the current p-XACT is reset. Additionally, if this mechanism were triggered by an external request, it would be acknowledged by the requestor.

5.3.4.2 Global Recovery

Given the fact that blocks are shared before consolidation, potential faults could be spread among cores. In case that a p-XACT is detected as faulty, the recovery mechanism is also responsible for notifying its consumers (including the lower p-XACTs of the same node). Thus, upon fault detection, the mechanism carries out different actions, depending on whether the affected p-XACT is either a consumer or a producer:

- **Consumer**. If the current p-XACT is a consumer, the produced values were not previously shared, therefore potential faults have not been spread outside the core. In this case, a local recovery of the current p-XACT is performed. If the recovery process is initiated by an external request, an ACK is sent back to the source of the request. Likewise, the mechanism is repeated for the upper p-XACT.
- **Producer**. In this case, the process sends a rollback request to all the consumers of the current p-XACT (indicated by its Producer Register). When all the ACKs are collected, a local recovery of the current p-XACT is initiated and this mechanism is repeated for the upper p-XACT.

The recovery process finishes when all the p-XACTs in a core have been recovered. As as a final step, the register checkpoint is written back to both master and slave, and the execution is resumed. Hence, on the one hand, the described method assures that, for a faulty core, a younger p-XACT is "undone" before an older one. On the other hand, consumers are restored before producers, in case of dependencies among different cores. We can see an example of a fault recovery in Figure 5.3. In this example, four cores C0, C1, C2 and C3 have executed four p-XACTs p0, p1, p2 and p3. In the figures, the data sharing is represented by a solid line and an arrow, while the order of precedence is indicated by a dotted line.

In (a), a fault is detected in p0 from C2, which initiates the recovery mechanism. In (b) the recovery mechanism proceeds with p3 from C2, the youngest p-XACT of the core. As p3 has no consumers, it is locally recovered. C2 repeats the same process with p2 and p1, which are also locally recovered. As p0 in C2 is producer, it cannot be recovered yet. In (c), C2 sends invalidations to its consumers C1 and C3 and waits for the corresponding ACKs. In (d), C1 performs the recovery for p3. As p3 is producer of C2, it sends a rollback request which is acknowledged by C2 since p3 has already been recovered. Then, p3 and p2 from C1 are rolled-back. Given the fact that p2 recovery was requested by C2, C1 informs it that the recovery for the affected p-XACT has been performed. In the same way, C3 recovers from p3, p2, p1 and p0. As p0 is the oldest p-XACT, the registered checkpoint is also recovered and finally, C3 acks C2. In (e), C2 has received all the ACKs, thus performing the rollback of p0 and restoring the backup of the register file. Meanwhile, C1 sends an abort request to C0 since p0 is its producer. C0 recovers from p3, p2, p1, p0, restores the register file checkpoint and acknowledges C1. Finally, in (f), C1 receives the acknowledgement from C0 and performs the rollback from p0 and recovers the register file backup.

5.4 Performance enhancements via Spatial Thread Decoupling

So far we have focused our discussion on the execution of redundant pair threads in 2-way SMT cores. The benefits of having both threads in the same core include a better use of the resources, the hiding of latencies (e.g., cache misses) and the avoidance of additional hardware. However, the major problem this design entails is performance degradation due to resource contention associated with SMT architectures. To avoid this penalty and provide a low-overhead solution, we propose to execute redundant threads in different individual (non-SMT) cores rather than in different SMT contexts.

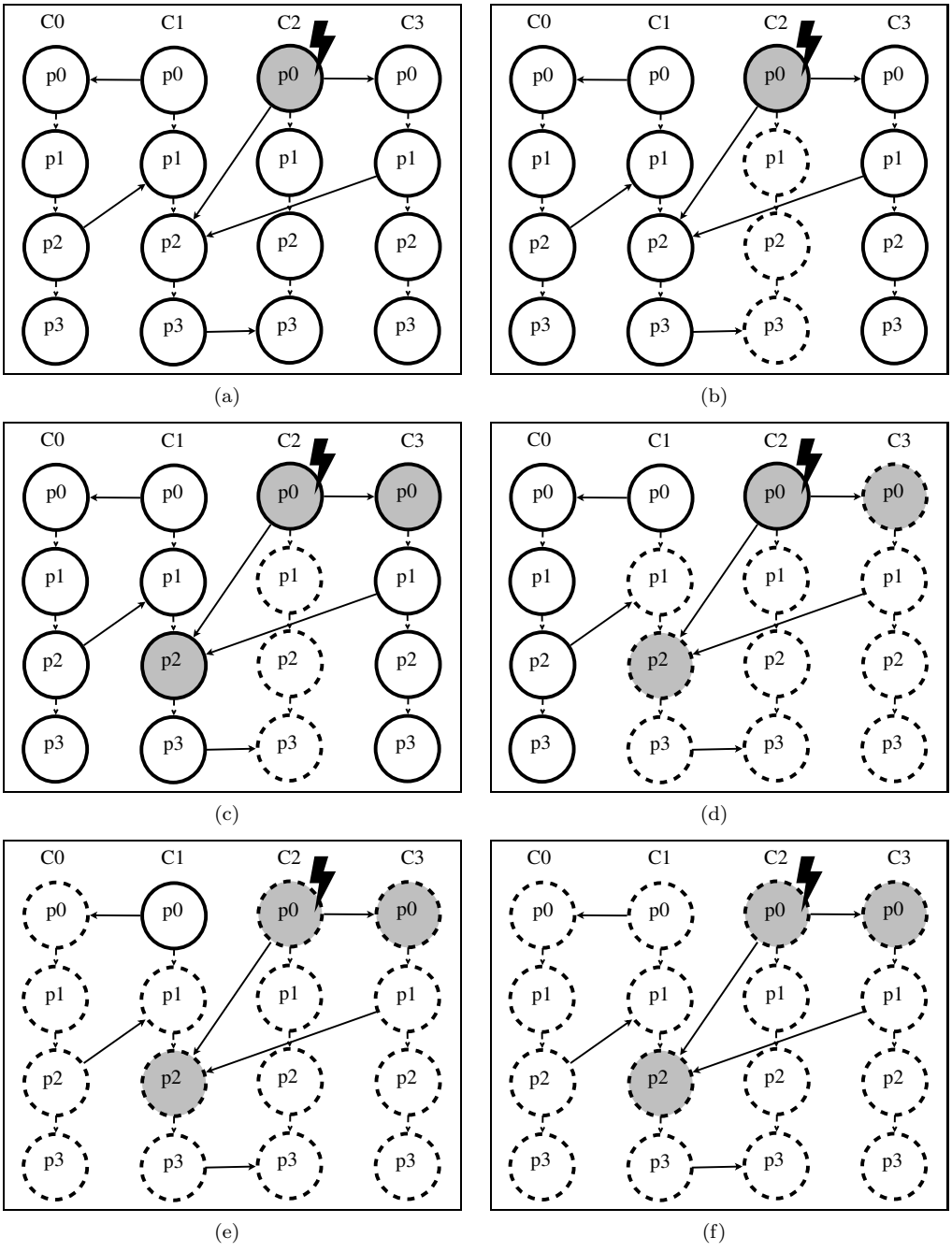


FIGURE 5.3: Fault recovery mechanism in LBRA.

5.4.1 Decoupling Thread Execution into Different Cores

The support for the execution of redundant threads in different cores is pretty straightforward given the software nature of threads. This support includes the creation of the redundant thread by the OS and the initialization of all the registers and related hardware structures including the log pointers.

However, the major drawback we face in this new scenario is the inter-core communication. Proposals like Reunion [83] rely on the use of dedicated fast lines to communicate redundant cores since the latency of the messages is crucial for its performance. Nonetheless, this measure is not desirable given the fact that it reduces the flexibility and adds a considerable hardware overhead.

Instead of that, our approach uses the interconnection network to communicate redundant cores. The major implication of this approach, thus, is the increase of the latency when accessing the log. This additional delay results from the travel across the chip of log blocks from master to slave cache. Eventually, if the latency is too high, the performance of the slave thread may be affected considerably. Nonetheless, this decreased slave thread performance slightly impacts on the master thread forward progress because of the temporal decoupling inherent to LBRA, which allows the master thread to commit several p-XACTs without the need of slave consolidations. If this buffering results insufficient, we can still adopt measures like allowing a higher number of in-flight p-XACTs, something which requires more hardware, or increasing the p-XACT size, which impacts directly on the total log size and reduces the effective capacity of the cache.

Leveraging coherence actions

However, as we will see in the evaluation Section, the overall performance of some applications in the spatially decoupled (non-SMT) environment results worse than in the coupled (using 2-way SMT cores) one. This behaviour is a consequence of the increased cache miss ratio of the master thread when accessing the log data blocks. In a coupled environment the log is allocated in the L1 cache and, virtually, all its accesses result in hits for both master and slave threads due to temporal locality. In a decoupled environment, though, the log blocks are written by the

master in its private cache and requested by the slave thread. This implies a cache-to-cache request and a transfer of the block permissions from M to S[3]. This extra latency only affects slave performance as we noted before. But the major problem results when the master thread eventually reuses a portion of the log. Since the last state of the log block is S, the master thread must re-acquire the write permissions, something which implies an invalidation message to the sharers (the slave thread in this case) and a subsequent acknowledgement. This results in an increase of log latency in the master which may affect the forward progress of the application.

In order to avoid this issue and given the singularity of this producer-consumer pattern, we have added a small hardware structure on the slave side together with the use of non-coherent requests to access the log. We call this structure the *log buffer*, a FIFO queue which stores log blocks. Slave accesses are performed through it. When the requested data is not present in the log buffer, the slave thread performs a cache-to-cache request which does not alter the coherence state of the memory subsystem. The size of the log buffer may be as small as a single block. However, being able to buffer multiple blocks brings on another opportunity. Given the fact that the log is accessed sequentially, a prefetch mechanism is very easy to implement. This can be used to obtain the log blocks the slave is going to use, hiding, most of the times, the latency when the slave accesses the log. Finally, note that master-slave consistency is assured since the blocks are frequently evicted from the log buffer because of its small capacity. This prevents the slave thread from reading an old version of a log block.

[3]Provided that the cache coherence protocol is MESI.

5.5 Evaluation

5.5.1 Simulation Environment

To evaluate the proposed LBRA architecture, we have simulated a tiled-CMP by means of Virtutech Simics [45] and GEMS [47]. Simics is a functional simulator executing a Solaris 10 Unix distribution simulating the UltraSPARC-III ISA. GEMS is a timing simulator which, coupled to Simics, supplies a hardware implementation of a transactional memory model called LogTM-SE [115].

TABLE 5.2: Simulation parameters.

16-way Tiled-CMP	
Processor Speed	2GHz
Memory and Cache	
Mem. Size	4GB
Mem. Latency	300 cycles
Cache Line Size	64 bytes
L1 cache	32KB, 1 cycle/hit
L2 cache	512KB/core, 15 cycles/hit
Network	
Topology	2D-Mesh
Protocol	MESI directory
Link latency	4 cycles
Flit Size	4 bytes
Link bandwidth	1 fit/cycle
LogTM-SE	
Signatures	Perfect
Log contents	Loads: data (4 bytes) Stores: data + address (8 bytes)

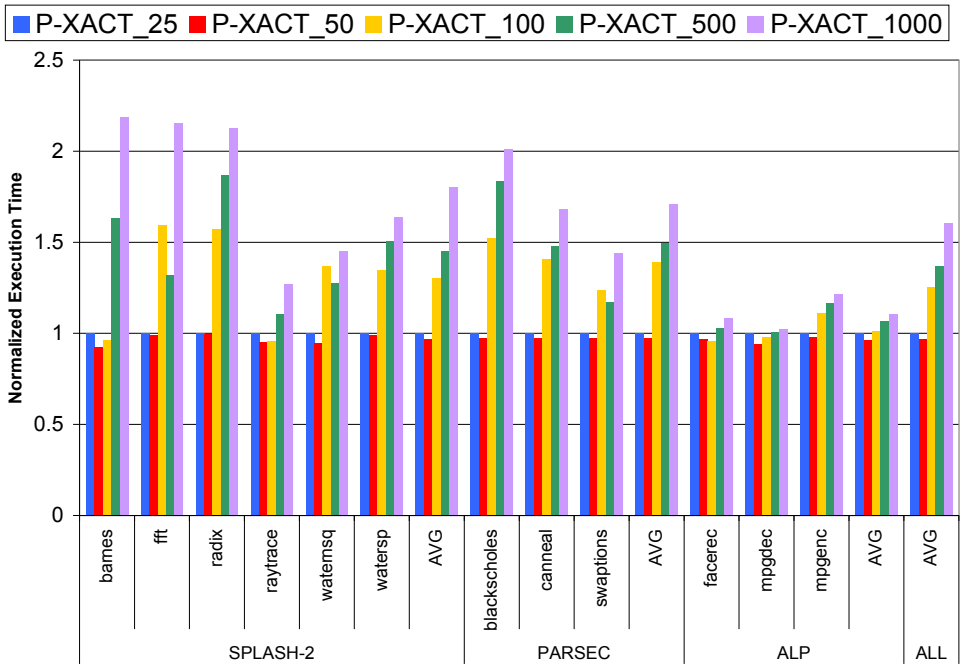
Table 5.2 shows the main parameters of the evaluated architecture. Each core of our 16-core CMP is a dual-threaded SMT with private L1 cache and a shared portion of the L2 cache. We conduct our experiments by executing several applications from SPLASH-2 [112] (barnes, fft, radix, raytrace, waternsq and watersp), ALPBench [43] (facerec, mpgdec and mpgenc) and PARSECv2.1 [7] (blackscholes,

canneal and swaptions) benchmark suites (refer to Chapter 3 for further details). The experimental results reported here correspond to the parallel phase of each program. Each experiment has been run with several random seeds as to take into account the variability of the multithreaded execution. Although LBRA allows the programmer to explicitly activate the redundancy in specific program parts, in this Chapter we assume full protection. Thus, every program instruction is redundantly executed.

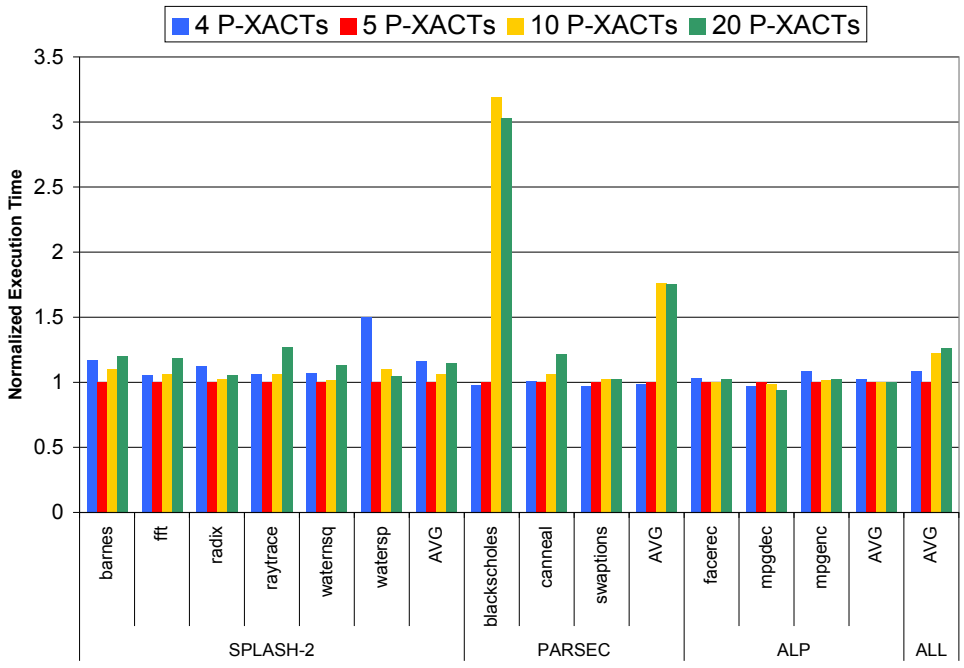
5.5.2 p-XACT Size Analysis

The size of a p-XACT is a key parameter in the architecture. A bigger size helps to increase the decoupling between master and slave threads. Unfortunately, this also increases the size of the log, incurring in a greater occupancy of the cache. Figure 5.4(a) shows a sensitivity analysis of the p-XACT base size in terms of memory instructions. The bars are normalized with respect to the case in which p-XACT length is 25 memory instructions. As we can see, decreasing p-XACT size from 1000 instructions to 100 instructions achieves performance gains. However, further decreasing the p-XACT size below 50 instructions it is not worthwhile. On average, 50-instruction size performs 3%, 2% and 3% better than 25 instruction size for SPLASH-2, PARSEC and ALP studied benchmarks, respectively. For smaller p-XACT sizes, the performance is even worse (not shown here for clarity), for the overhead incurred in every p-XACT creation (register initializations mostly) becomes more valuable.

Another interesting parameter is the maximum number of p-XACTs which the master can commit without consolidation. At one end, a higher number of in-flight p-XACTs facilitates decoupling, but it also adds more hardware as illustrated in Figure 5.1. In addition, the size of the log grows, increasing thus the cache miss ratio of the architecture. At the other end, if the number of in-flight p-XACTs is low, in situations in which the slave thread is unable to keep up with the master (because of dependencies in consolidations, for example), this turns into a bottleneck since the master must be stalled. This behaviour can be observed in Figure 5.4(b), which shows the execution time normalized with respect to using



(a)



(b)

FIGURE 5.4: Sensitivity analysis for p-XACT size and number of in-flight p-XACTs.

5 p-XACTs. For 4 in-flight p-XACTs, the stalls of the master execution are responsible for a performance degradation of 16% in SPLASH-2 and 2% for ALP (almost no degradation for PARSEC benchmarks) in relation to 5 in-flight p-XACTs, which is the best configuration for the studied benchmarks. For a higher number of in-flight p-XACTs, the overhead specially increases in benchmarks such as blackscholes and canneal, in which the cache miss ratio raises significantly.

5.5.3 Overhead of the Fault-Free Case

In this Section we compare LBRA architecture with a base case composed by a 16-core CMP running the 16-threaded applications mentioned in Section 5.5.1. We quantify the performance in a fault-free scenario, which is usually considered as the common case. Although LBRA allows the programmer to select specific program parts to protect while leaving the rest unprotected, for this evaluation, we provide redundant execution for all the program instructions.

Three different factors are responsible for the performance degradation of LBRA. First and foremost, the cost of redundancy itself (note that the use of dual SMT cores aggravates this performance degradation as a result of the higher resource contention of master-slave pair threads). Second, the capacity of the L1 cache, which is reduced because of the the log used to bypass data between master and slave threads and to provide a backup. For this reason, smaller p-XACTs normally achieve better performance. And finally, the stalls in the consolidation phase due to dependencies among two or more p-XACTs. Fortunately, these consolidation stalls are uncommon (virtually non existent). Furthermore, the master thread is rarely stalled as a result of the proposed mechanism which allows to execute several p-XACTs without consolidation.

Figure 5.5 depicts the behaviour of the coupled LBRA approach labeled as LBRA_C (the first version of LBRA proposed in Section 5.3). As we can see, the performance degradation in our first approach ranges between 38% (facerec) and 16% (radix) with an average of 24%. Although the experimented degradation is noticeable across all the studied benchmarks, it is worth noting the impact on ALP benchmarks because of the inherent SMT degradation, something which

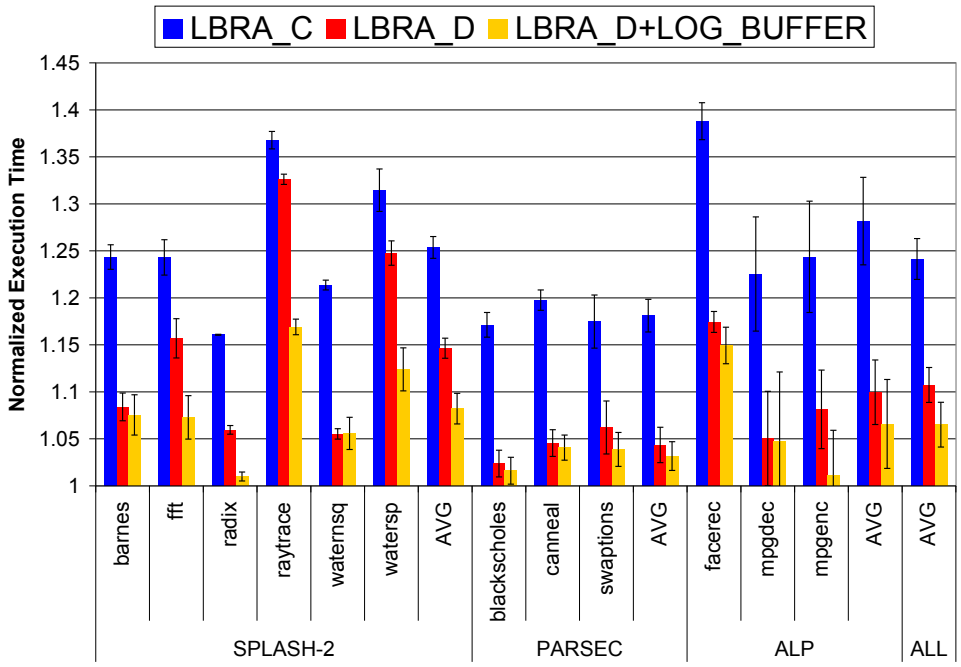
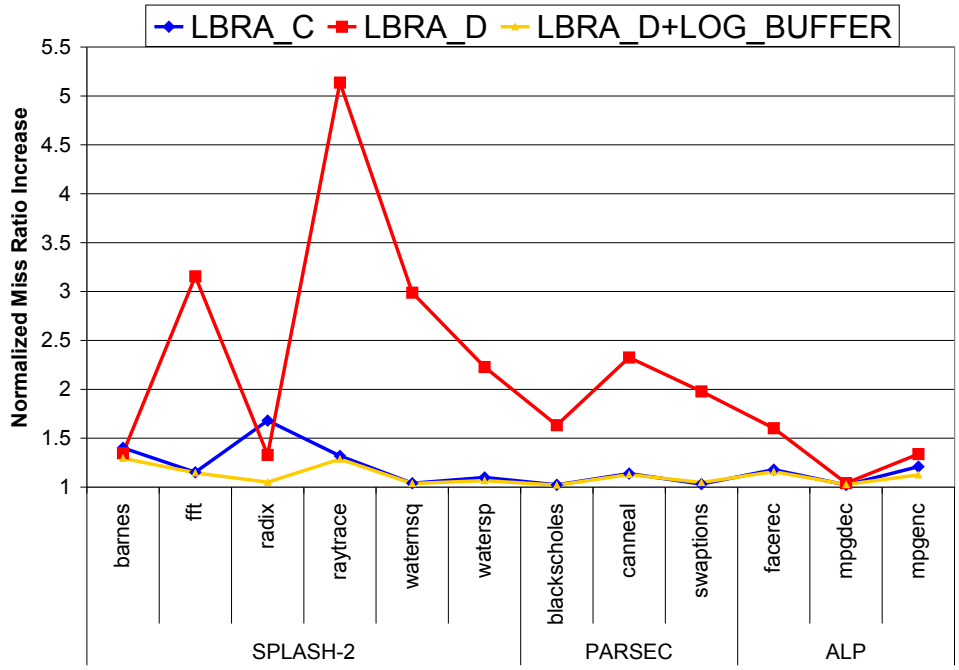


FIGURE 5.5: LBRA performance in a fault-free scenario.

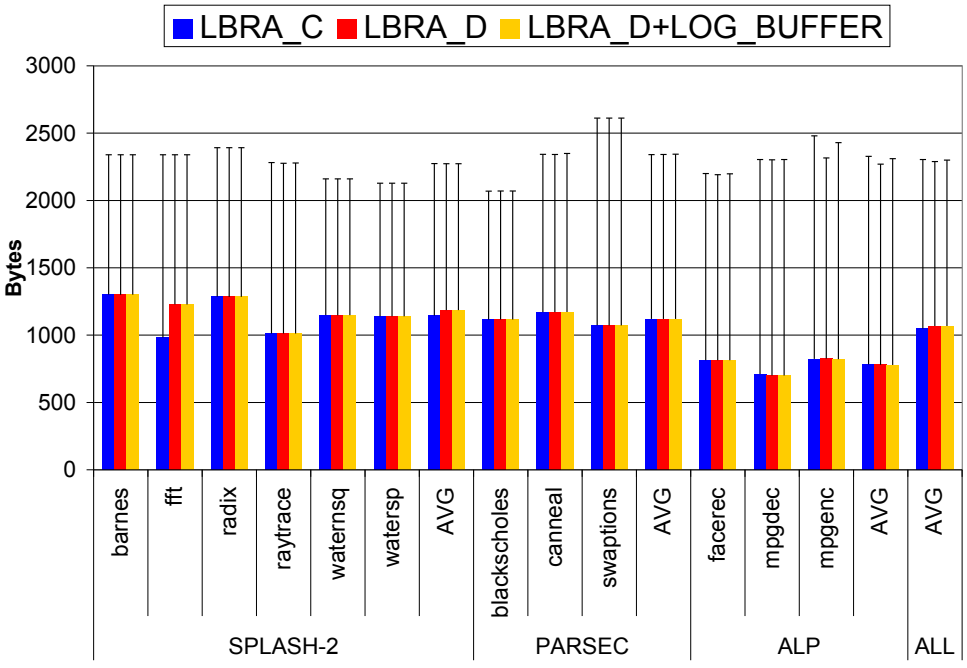
is consistent with related studies such as Sasanka *et al.* [80], with claim a better performance of CMP architectures compared to SMT architectures for multimedia workloads.

To overcome this issue, in Section 5.4 we proposed to decouple redundant threads in different non-SMT cores. Thus, we avoid degradation due to resource sharing but we increase the hardware costs. Specifically, the number of cores increases to 32 distributed in a 4x8 2D-mesh in which master and slave cores are placed at a 1-link distance. Figure 5.5 depicts the overhead of this approach, labeled as LBRA_D which ranges from 32% (raytrace) to 2% (blackscholes) with an average of 11%, clearly outperforming LBRA_C (24% on average). But, besides the hardware overhead, LBRA_D results in a noticeable increase of the L1 cache miss ratio as we can see in Figure 5.6(a). However, the size of the log remains constant with an average of 1KB (up to 2.4KB) for all the studied benchmarks as we can see in Figure 5.6(b). Thus, it is clear that the miss ratio increase is due to the conflicts between master and slave cores when accessing the log.

To reduce the miss ratio in LBRA_D we incorporate to the architecture a log



(a) LBRA L1 miss rate increase.



(b) LBRA average and maximum log size.

FIGURE 5.6: LBRA miss rate and log size.

buffer which is used by the slave thread to obtain log blocks without affecting their coherence state, as explained in Section 5.4. The capacity of the log buffer can be as small as 1 entry. Nonetheless, our experimental analysis shows that with a capacity of 3 blocks, we obtain the optimum performance by using a simple prefetch mechanism. The prefetch strategy we follow consists of requesting the next logical log block whenever there is, at least, one free entry in the buffer. This approach is labeled in Figure 5.5 as LBRA_D+LOG_BUFFER. In this case, the time overhead is reduced to the range between 17% (raytrace) and 1% (radix) with an average of 6.5% for all the studied benchmarks. The performance improvement in comparison to LBRA_D is easily explained due to the reduction in the cache misses, as we can see in Figure 5.6(a). In conclusion, the benefit of LBRA_D+LOG_BUFFER is twofold. First, we avoid performance degradation thanks to the redundant threads in the same SMT core. And second, we decrease the impact over the L1 miss ratio by leveraging the coherence for log blocks.

5.5.4 Comparison Against Previous Work

In this Section, we evaluate the performance impact of previous approaches in a common framework, i.e. a direct-network environment and using 2-way SMT core redundancy or simply core redundancy.

Whereas LBRA_C uses the cache to communicate log values between redundant threads, REPAS (as proposed in Chapter 4) uses specialized hardware, something which imposes a considerable complexity overhead. Besides, as we can see in Figure 5.7, LBRA_C reduces the execution time overhead of REPAS for the majority of studied benchmarks with a 8% on average with respect to the base case. For applications that lay more pressure over the LVQ and SVQ queues such as watersp and swaptions, REPAS incurs in noticeable overheads, since the leading thread must be stalled until resources are available. LBRA_C does not suffer from this problem since all the data communication is produced through the cache.

In the same Figure 5.7, we can see the execution time overhead of DCC and LBRA_D implementing the log buffer. In these two cases, redundant threads are executed in different cores, eliminating thus the degradation inherent to the SMT

execution. The counterpart, however, is that the number of cores is doubled. The consistency mechanism is the major degradation source in DCC, as we explained in Section 4.2.1, increasing time overhead in 31% on average with respect to the base case.

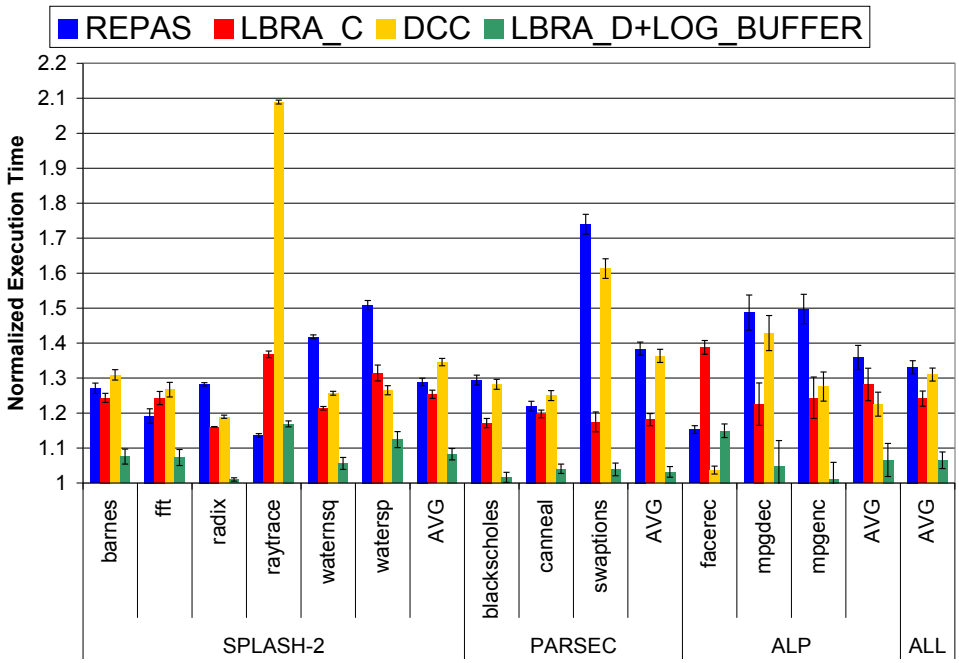


FIGURE 5.7: Performance comparison of LBRA versus REPAS and DCC.

Finally, in Figure 5.8 we can see the execution time overhead for LBRA_D (with the log buffer), REPAS and DCC under different fault rates expressed in number of faults per million of execution cycles. Since realistic rates barely affect the performance of mechanisms, in this experiment we have used rates which are higher than those expected in a real scenario so as to show the kindness of the different approaches. The time to recover from a fault depends on the speed to detect the fault and to undo all the potentially affected work. REPAS is the fastest mechanism to detect and correct an error because of the small delay between redundant threads (less than 200 cycles). On the contrary, DCC spends roughly 10,000 cycles to recover because of its long checkpoint intervals. The overhead of LBRA_D is between the other two. As depicted in Figure 5.8, while the fault rate is less than 10 faults per million of cycles, LBRA_D is still the most suitable approach but, whenever the fault rate increases to 100, the small overhead introduced by

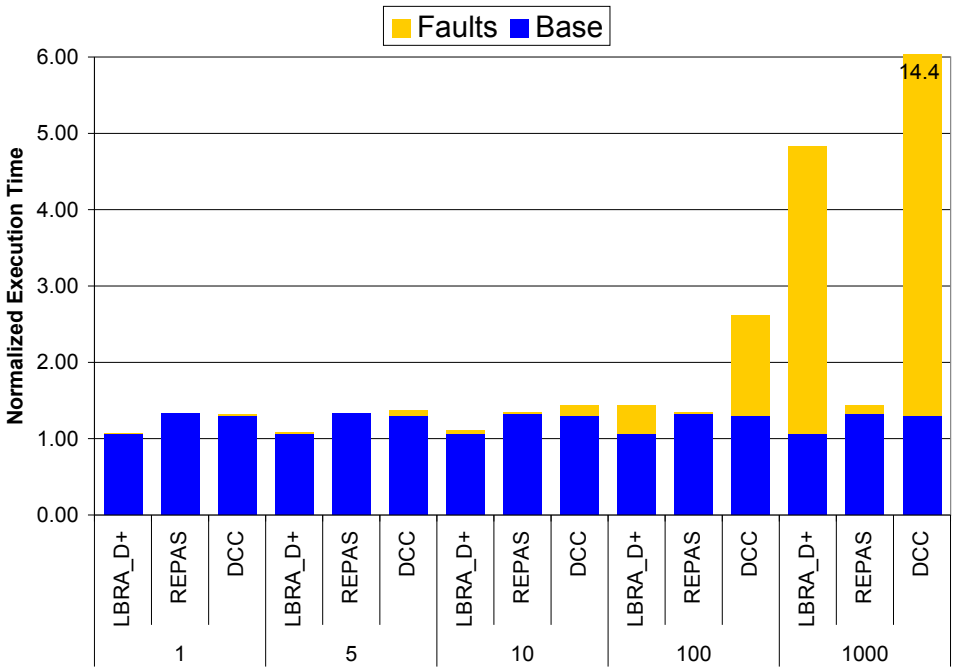


FIGURE 5.8: Execution time overhead for several fault rates.

REPAS makes it the best choice. DCC's performance is always the less advisable in a faulty environment as a result of the long recovery latency caused by its long checkpoint intervals.

To sum up, we have shown that both LBRA_C and LBRA_D are able to outperform previous approaches in the same environment with several advantages. First, we avoid the use of queues to bypass data values between threads as in REPAS, something which augments considerably the complexity of the design. Second, the modifications over the memory system are minimum whereas DCC induces virtually to the creation of a specific and new coherence protocol [37] and requires special measures to deal with master-slave inconsistencies. In the same way, in a faulty environment with realistic fault rates, LBRA_D is also the most adequate approach in terms of performance degradation.

5.6 Concluding Remarks

CMOS scaling exacerbates hardware errors making reliability a considerable concern for present and future microarchitecture designs. However, mechanisms to provide fault tolerance in architectures must accomplish several objectives such as low performance degradation, energy consumption and area overhead.

In this Chapter we have presented a novel low-overhead mechanism to deal with transient faults in present and future architectures. To this end, we introduce LBRA, an architecture design based upon LogTM-SE, a well-established hardware implementation of Transactional Memory. LBRA executes redundant threads which communicate through a virtual memory log placed in cache. The goal of the log is twofold. First, it provides input replication for both threads and, second, it is used to recover the architectural state of the system after a fault is detected.

LBRA main features include: a) a consistent view of the memory between master and slave thread, avoiding input incoherences; b) both transient fault detection and recovery; c) more scalability and higher decoupling than previous proposals; d) low-performance overhead.

LBRA is presented in two flavours. In the first approach, redundant threads are executed in the same dual-threaded SMT core. This provides a low-hardware overhead but imposes a noticeable performance degradation as a counterpart. To solve this issue, we have proposed a second approach in which redundant threads are executed in different cores, increasing, thus, the hardware requirements. To address the inter-core communication latency, we rely on the use of a simple yet effective mechanism comprised of a log buffer, a prefetch strategy and slight modifications of specific coherence actions.

We have compared and evaluated the proposed designs using full system simulation to measure the performance degradation in a fault-free environment with parallel benchmarks. We have shown that our proposals address the fault-tolerant goal imposing 24% and 7% execution time overhead for the coupled and decoupled mechanism, respectively. Furthermore, we have shown that LBRA presents

several advantages with respect to state-of-the-art approaches that we have also evaluated in the same framework.

Chapter 6

Modelling Permanent Fault Impact on Cache Performance

SUMMARY:

The traditional performance-power benefits enjoyed for decades from technology scaling are being challenged by reliability constraints. Increases in static and dynamic variations have led to higher probability of parametric and wear-out failures. This is particularly true for caches that dominate the area of modern processors and are built with minimum sized but prone to failure SRAM cells. It is, therefore, imperative to develop effective methodologies that facilitate the exploration of reliability techniques for processor caches.

Our attempt, presented in this Chapter, to address the cache reliability challenge is an analytical model for studying the implications of block-disabling approaches due to random cell failure on the cache miss rate behaviour. The proposed model is distinct from previous works in that is an exact model rather than an approximation and yet it is simpler than previous methodologies. Its simplicity stems from the lack of the use of fault-maps in the analysis. As far as we know, all previous related works rely on the continued use of some number of fault-maps which is not justified why it is representative of the expected behaviour, and therefore, it questions the generality of the conclusions of such studies.

The analytical model capabilities are illustrated through a study of the expected and standard deviation of the miss-rate trends in future technology nodes. The model is also used to determine the accuracy of the random fault map methodology. The analysis reveals, for the assumptions, programs and cache configuration used in this study, a surprising result: a relative small number of fault maps, 100-1000, is sufficient to provide accurate average and standard deviation estimations. Additional investigation revealed that the causes of this behaviour is the high correlation between the number of accesses and the access distribution between cache sets.

6.1 Introduction

For the past 50 years, technological advances have enabled the continuous miniaturization of circuits and wires. The increasing device density offers designers the opportunity to place more functionality per unit area and, in recent years, has allowed the integration of large caches and many cores into the same chip. Unfortunately, the scaling of device area has been accompanied by at least two negative consequences: a slowdown of both voltage scaling and frequency increase due to slower scaling of leakage current as compared to area scaling [9, 22, 101], and a shift to probabilistic design and less reliable silicon primitives due to static [10] and dynamic [11] variations.

These alarming trends are leading to forecast that the performance and cost benefits from area scaling will be hindered unless scalable techniques are developed to address the power and reliability challenges. In particular, it will be impossible to operate all on-chip resources (even at the minimum voltage for safe operation) due to power constraints, and/or the growing design and operational margins used to provide silicon primitives with resiliency against variations which may consume the scaling benefits.

A recently published resilience roadmap underlines the magnitude of the problem we are confronting [57]. Table 6.1 shows the p_{fail} (probability of failure) predicted in [57] for inverters, latches and SRAM cells due to random dopant fluctuations as a function of technology node (the trends for negative-bias-temperature-instability are similar [116]). The trends clearly show that for all types of circuits the p_{fail} increases at a much faster rate than the scaling rate. However, not all circuits are equally vulnerable; SRAM cells which are usually built with minimum sized devices are more likely to fail. Furthermore, if we resort to voltage operation below safety margins the SRAM p_{fail} increases exponentially [111].

Technology	**Inverter**	**Latch**	**SRAM**
45nm	≈ 0	≈ 0	6.1e-13
32nm	≈ 0	1.8e-44	7.3e-09
22nm	≈ 0	5.5e-18	1.5e-06
16nm	2.4e-58	5.4e-10	5.5e-05
12nm	1.2e-39	3.6e-07	2.6e-04

TABLE 6.1: Predicted p_{fail} for different types of circuits and technologies [57].

These trends render essential the development of reliability techniques for future processors which are both scalable and performance-effective. This is especially important for caches that take most of the real-estate in processors and contain numerous, vulnerable-to-failure, SRAM cells.

In the past, when variations issues were less dominant, it may have been acceptable during post-manufacturing tests to discard chips even with a single faulty cache bit. Nowadays, this is not a viable approach as reflected by the use of extensive spare columns and rows in contemporary cache SRAM arrays [39]. However, the amount of spares needed to ensure a fault-free cache can grow faster than area scaling rate and, consequently, diminish scaling benefits. An approach that can reduce the amount of spares needed, to address parametric variations and power constraints, is the use of more robust cells [2]. These cells have lower p_{fail} but each typically require more transistors and larger area, which result in longer overall cache access time. Another approach is frequency and voltage binning that results in operating a chip at lower frequency or higher voltage than intended, so that all cells can be accessed correctly [10]. This ensures functional correctness but it either reduces performance or increases power/temperature.

The approaches above aim at providing fault-free caches by mitigating manufacturing and static parametric faults. This is desirable but pretty unrealistic to accomplish cost-effectively for future caches due to the mismatch between the cell p_{fail} versus area rate of scaling. Furthermore, wear-out faults that occur in the field, are becoming more common [57]. Consequently, we may be forced to ease the requirement of shipping only chips with fault-free caches and replacing parts that experience a wear-out fault. But this requires performance-cost effective mechanisms to deal with permanent faults in a cache during operation.

One inexpensive option is to rely on the error-correction-codes (ECC) already in place to detect and correct soft-errors. However, ECC is not a performance friendly mechanism for permanent errors because, potentially, every access to a faulty block will incur in the ECC repair overhead, e.g., for a typical SEC-DED implementation the decoding process latency is around 2x times the encoding one [56]. Furthermore, ECC soft-error capabilities are reduced when some bits protected by the ECC code are already faulty. Finally, the more faults, the stronger the ECC code required, and its overhead. Thus, ECC may not be the best option to repair permanent or wear-out faults for caches.

Another approach is to disable blocks [64, 86] which contain faulty bits upon permanent error detection (at manufacturing time or in the field). Such disabled blocks are not replaced with a spare[1]. Therefore, the cache capacity is reduced. Block disabling is an attractive option because it has low overhead, e.g. 1 bit per cache block[2], but its reduced cache capacity may degrade performance. Therefore, it is important to determine the performance implications of block disabling to assess its usefulness.

Block disabling is not a new concept. It has been proposed and evaluated before [31, 40, 41, 65, 75, 81], for example, as a way to improve manufacturing yield [41, 65, 86], and to enable cost-effective operation below Vcc-min [36].

All these previous disabling-based studies rely on on a specific number (small or large) of random fault-maps. The random fault-maps indicate the location of

[1]Disabling can be employed after spares have been exhausted.

[2]This logical bit needs to be resilient either through circuit design or extra protection because, if faulty, it would render the whole cache faulty.

faulty cache cells and determine the disabled faulty cache blocks. They are used to either obtain the performance degradation of a program through cycle-accurate simulation or to determine the impact on miss-rate of a program's address trace through an analytical model proposed by [65]. We claim that all these studies are limited because they do not provide a justification as to why the particular number of fault maps they use, which is typically a very small subset of the actual number of possible mappings, is representative of the expected behaviour. Therefore, it remains unclear whether the conclusions reached in these studies are representative.

Our proposition to address this shortcoming is an analytical model that calculates the Expected Miss Ratio (EMR) for a given application memory address trace, along with a cache configuration and random probability of permanent cell failure (p_{fail}). Furthermore, we show how to obtain the standard deviation for the EMR (SD_MR) which provides an indication for the range of expected degradation of the cache. Finally, we explain how to produce a probability distribution for the EMR for a given number of faulty blocks. All this is accomplished without producing or using any fault-maps.

Our model capabilities are demonstrated through an analysis of the trends of the mean and standard deviation of the cache miss rate with smaller feature size (and p_{fail}) for L1 and L2 cache. This analysis reveals that, for the programs and cache configurations used in this study, the random fault methodology provides highly accurate mean and standard deviation estimations with 1000 maps for the L1 cache and with 100 for the L2 cache. These surprising results are investigated through correlation analysis revealing a very high degree of correlation between the number of accesses and the access distribution across sets which mean that a relative small number of fault maps is sufficient to capture the mean and standard deviation of the cache miss rate.

Furthermore, we study the EMR, access time, power, and area trade-offs of different cache architectures. Finally, we compare block-disabling with word-disabling, showing that, for higher p_{fails}, word-disabling is more adequate due to its reduced EMR, although with the mayor drawback of increasing cache latency access.

The remainder of this Chapter is organized as follows: Section 6.2 reviews related work. Section 6.3 presents our model to calculate the EMR and SD_MR. In Section 6.4 we describe the methodology we follow in the evaluation Section which is presented in Section 6.5. Finally, Section 6.6 summarizes the main conclusions of this work.

6.2 Related Work

There have been several previous proposals studying the impact of permanent faults on caches and the ways to mitigate them.

Sohi [86] studied the performance implications of caches with disabled portions such as blocks and sets. In that work, the expected miss-rate increase is obtained using several random fault-maps which are evaluated using trace-driven simulation.

Pour and Hill [65] extended Sohi's work with a more accurate methodology by introducing the use of the all-associativity simulation to determine within a single run all possible access patterns for an address trace over a cache. In fact, this study provides an expectation for the miss ratio of faulty caches in a similar way as our model does. However, the problem they try to solve in [65] is to estimate the EMR for a fixed number of faults. For that, they generate all the possible distributions of these faults over the cache sets. This is a well known problem called *partition problem* which falls within the category of NP-complete problems. In this Chapter, we present an analytical model, showing that the methodology in [65] can be applied to the study of the EMR and SD_MR for a given p_{fail} without the need to generate fault maps. Furthermore, we also show that we can approximate a fixed number of faults with a p_{fail} analytically and generate probability distributions for the EMR.

Lee *et al.* [41] study the performance impact on IPC and miss ratio of faulty caches. However, their study still relies on the execution of random maps which are generated by means of Monte Carlo method. Finally, there are several other studies [1, 31, 36, 75, 81] studying the impact of faults over caches using random maps.

6.3 Analytical Model for Cache Miss Rate Behaviour with Faults

In this Section, we present an analytical model that can determine the expected Miss Ratio (EMR), standard deviation of the Miss Ratio (SD_MR), and a probability distribution of miss-ratios (PD_MR) for a given program address trace, cache configuration and random probability of permanent cell failure (p_{fail}). The EMR captures the average performance degradation due to random faulty cells. The SD_MR provides indication about the range of the performance degradation, whereas PD_MR reveals the shape (distribution) of the performance degradation. These characteristics can be used to assess the implications of faults in a cache and compare different cache reliability schemes.

The model key novelty is that it does not rely on fault-maps and is exact rather than an approximation, i.e., it determines the above as if *all* possible fault-maps for a given random p_{fail} have been considered. Previous studies that relied on fault-maps may not have produced representative conclusions because they can not generate and evaluate, in general, all possible cache fault-maps for a given p_{fail} in a reasonable amount of time. We investigate the accuracy of the random fault-map methodology in Section 6.5.2.

6.3.1 Assumptions and Definitions

The model assumes that the permanent faulty cells occur randomly (uncorrelated) with probability p_{fail}. This random fault behaviour is indicative of faults due to random-dopant-fluctuations and line-edge-roughness, two prevalent sources of static variations.

A cache configuration is defined by the number of sets (s), ways per set (n), and block size in bits (k). We consider a block containing one or more permanent faulty bits as faulty. In that case, the faulty block is disabled and reduces the capacity of the cache. The faults are assumed to be detected with post-manufacturing and boot time tests, ECC, and built in self tests.

The model is suited for any deterministic replacement policy. However, without loss of generality we have focused on a basic LRU policy.

Each program address trace is simulated through a cache simulator to obtain for a given cache configuration the vector M. This vector contains $n+1$ elements, one element more than the number of cache ways. M_i corresponds to the total misses when there are only $n-i$ valid ways in each set in the cache. More specifically, M_i equals to the sum of all references that hit on the i least recently used blocks in each set, plus the misses of the fault free cache. For example, M_0 equals to the misses of a fault-free cache; M_n represents the misses of a cache where all ways are entirely faulty, meaning that all accesses are misses; and M_1 equals to the misses of the fault-free plus all the hits in the LRU position.

6.3.2 EMR and SD_MR

This Section shows how the model obtains the EMR and SD_MR given a cell p_{fail}, cache configuration and the miss vector of an address trace. The model obtains the probability for a cache block failure using the following expression (based on well known binomial probability):

$$p_{bf} = 1 - (1 - p_{fail})^k \tag{6.1}$$

Although p_{bf} gives information about the fraction of blocks that are expected to fail in the cache, the impact on the miss ratio is unknown as it depends on the fault location and amount of accesses that map faulty block locations. However, with the p_{bf} we can obtain the probability distribution pe_i for the number of faulty ways in a set:

$$pe_i = \binom{n}{i} p_{bf}^i (1 - p_{bf})^{n-i} \tag{6.2}$$

which provides, for every possible value of i $[0...n]$, the probability of having $n-i$ non-faulty ways. This distribution is very useful because it provides the

complete picture about how likely to loose a given number of ways is in a set, and what is more, it can be used to obtain the expected number of misses.

The expectation of a random variable $X = x_0, x_1..., x_m$ for which each possible value has probability $p = p_0, p_1, ..., p_m$ can be calculated as:

$$E[X] = \sum_{i=0}^{m} x_i \cdot p_i \tag{6.3}$$

In our case, the random variable X corresponds to the total number of misses for a cache with faults, x_i to the total misses when there are only $n - i$ valid ways in each set in the cache, and p_i the probability of having i faulty ways in a set. Therefore, we can express the expectation of the number of misses of cache with disabled blocks as:

$$E_{misses} = \sum_{i=0}^{n} M_i \cdot pe_i \tag{6.4}$$

and obtain the expected miss ratio of the cache using:

$$EMR = \frac{E_{misses}}{accesses} \tag{6.5}$$

This simple formula can be used to obtain the exact EMR without using fault-maps. The key insight behind this formula, expressed better in Eq. 6.2, is that caches have a useful property: for the same number of faulty blocks f in a set the reduced associativity will be the same $n - f$. I.e., for analyzing block-disabling approaches, what matters is the number of faulty-ways in a set, not which specific ways in the set are faulty. As a result, this reduces the complexity problem.

The EMR provides a useful indication of the average case performance for a given p_{fail}. However, we have no information about the variation in the miss ratio. Variation information is useful for assessing whether disabled blocks lead to caches with wide variation (less predictable) miss rate.

One way to measure this variation is through the standard deviation of the miss ratio or SD_MR. Unfortunately, the standard deviation cannot be directly obtained for the whole cache. However, given that we already know the probability distribution of faulty blocks in a set, we can calculate its variance as follows:

$$\forall j[0...s], VAR\_E_{misses_j} = \sum_{i=0}^{n} pe_i \cdot (x_{ij} - E_{misses_j})^2 \tag{6.6}$$

where x_{ij} is the number of misses obtained when having $n-i$ non-faulty ways in the j_{th} set.

Although the total EMR is equal to the sum of individual sets EMR_j:

$$EMR = \sum_{j=1}^{s} EMR_j \tag{6.7}$$

we cannot combine the variation of each set in the same way. Instead, we compute the deviation for the misses of the whole cache SD_MR by using the root mean square in the form:

$$SD\_MR = \frac{\sqrt{\sum_{j=1}^{s} VAR\_EMR_j}}{accesses} \tag{6.8}$$

6.3.3 EMR Probability Distribution

The SD_MR only provides the range of deviation of the expected miss ratio (EMR). However, it may also be useful to know the probability distribution of cache misses (PD_MR) within the deviation range.

We propose to build a probability distribution of misses in a stepwise manner. We first calculate the EMR for every possible number of faulty blocks (0 to the number of cache blocks), and then combine this information with the probability of that given number of faulty blocks to occur.

Equation 6.9, similar to Equation 6.2, gives the probability of x number of faulty blocks, for a given block probability failure:

$$\binom{s \cdot n}{x} p_{bf}^{x} (1 - p_{bf})^{s \cdot n - x} \tag{6.9}$$

This equation can be evaluated for different x to obtain a probability distribution. Then, we need to calculate the EMR for every possible number of faults. This problem has traditionally been solved by means of random fault maps [65].

For a given number of faults this problem is analogous to selecting at random n balls from an urn that contains dk balls without replacement, where d is the number of unique colours and k is the number of balls of each color. The urn represents the cache, the variable n the faults, d is the number of blocks and k the number of bits in each block. The mean number of distinct blocks, u, that contain at least one faulty cell in a cache with n faulty cells can be calculated with very high accuracy [114]:

$$u = d - d(1 - p_{fail})^{k} \tag{6.10}$$

This means that we can obtain the PD_MR analytically, without fault-maps. Simply use Equation 6.10 to convert the number of faulty blocks to p_{fail}. This gives the expression:

$$p_{fail_i} = 1 - \sqrt[k]{\frac{s \cdot n - x_i}{s \cdot n}} \tag{6.11}$$

This way, what p_{fail} results in x_i faults in the cache is calculated. Then, every p_{fail_i} can be used to calculate the EMR associated to each number of faulty blocks.

6.4 Methodology

We have shown in Section 6.3 how our model for estimating the EMR works. In this Section we explain the methodology we use to extract the behaviour of applications to test our model and how to produce random fault maps.

6.4.1 Generating Maps of Accesses

The input to our analytical model is a map of accesses to the cache for every application. With no additional support, we would not need to run the application for every possible cache configuration. In order to avoid this cost, we have used an algorithm called all-associativity simulation [29], previously used in [65].

This algorithm takes as input a trace of memory accesses which is converted to a map of accesses to a cache of any desired configuration (sets and ways) following a deterministic replacement policy (LRU in our case). This allows us to obtain the number of accesses per way and per set without the need of new simulation runs. The output of the algorithm is a matrix in which each row corresponds to a set and each column to a position in the LRU sequence. Each value of the matrix indicates the number of accesses to every position in the LRU sequence for every set. This information is highly useful for offline analysis given that we can extract the number of misses for a given number of ways w in our cache by simply adding the accesses for the last $n - w$ columns of the matrix.

We can see an example of how this mechanism works in Table 6.2 for a cache with 4 ways and 4 sets. The first column from the left (*way3*) refers to the accesses to the first position in the LRU sequence for each set. Last column (*cold misses*) refers to accesses to data not previously accessed or with a position in the LRU sequence greater than our threshold, meaning a capacity miss. In order to calculate the number of misses of the cache in a fault-free environment, we simply need to sum the accesses which appear in the last column. The number of misses of this example would be 570 $(100 + 90 + 200 + 180)$.

(a)

	way3	way2	way1	way0	cold misses
set0	120	100	110	60	100
set1	150	140	110	55	90
set2	180	134	80	50	200
set3	220	200	100	30	180

(b)

	way3	way2	way1	way0	cold misses
set0	490	370	270	160	100
set1	545	395	255	145	90
set2	644	464	330	250	200
set3	730	510	310	210	180

(c)

	way3	way2	way1	way0	cold misses
cache	2409	1739	1165	765	570

TABLE 6.2: EMR calculation after all-associativity algorithm execution.

But we can also use this table to compute the misses in a faulty environment. For this, and according to our model formulated in Equation 6.5, we need to calculate the number of misses we will obtain if a given number of ways (from 0 to 4 in our example) were disabled in the cache due to permanent faults. First, we accumulate the number of accesses in every position per set. The result of this is Table 6.1(b). Finally, we perform the same operation per set to get the vector in Table 6.1(c). This vector indicates exactly the number of misses our cache would suffer as a result of losing from 0 to w ways. Finally we can use this vector to apply our model and get the EMR and SD_EMR for a faulty cache.

6.4.2 Random Fault-Maps

A way to compare our model is by means of random maps of faults. Each one of these maps reflect a number of disabled ways for every set in our architecture, regardless of the location of these faults in the set. This is so because the position does not alter the number of misses in the cache because of associativity.

To generate these random maps we have used the GNU Scientific Library (GSL). We generate random numbers in the interval [0,1] to mark the faulty blocks in terms of a given p_{fail}. Notice that we are not interested in the position of the block in the set but in the location of these blocks across all sets. Finally, we compute the number of misses produced by a faulty map by using the access maps obtained with the all-associativity algorithm.

6.5 Evaluation

For our experiments, we have simulated a processor architecture by means of Virtutech Simics [45] and GEMS [47]. We have performed several modifications to the simulator in order to extract memory address traces. Then, we have used these traces to generate the map of accesses for every possible cache configuration by means of the all-associativity algorithm as explained in Section 6.4.

The studied benchmarks and input sizes are reflected in Table 6.3. As we can see, we conduct our experiments by executing both sequential applications from SPECcpu-2000 [28], and parallel applications from SPLASH-2 [112] and PARSECv2.1 [7]. In the case of parallel benchmarks, the experimental results reported here correspond to the parallel phase of each program. Benchmarks are executed in a CMP with private L1 caches and a shared L2. The number of cores in each case depends on the number of threads. In all cases, the warming up of caches has been taken into account. For the evaluation of L1 caches, benchmarks have been executed for 1 billion of cycles, whereas for the evaluation of the L2 cache, benchmarks have been run to completion.

The different p_{fails} used for the evaluation of the caches are depicted in Section 6.1 with the exception of the 6.1e-13 p_{fail}, which produces virtually no faulty blocks in our experiments. All these p_{fails} are predictions for the fault probability of SRAM cells for different scales of integration. Additionally, we have used other p_{fail} such as 1e-03, which is usually studied in related works.

SPEC	**bzip2**	reference
	gap	test
	gzip	reference
	parser	test
	twolf	test
	vpr	test
SPLASH-2	**Barnes**	8192 bodies, 4 time steps
	Cholesky	tk16.O
	EM3D	38240 objects
	FFT	256K complex doubles
	Ocean	258x258 ocean
	Radix	1M keys, 1024 radix
	Raytrace	10Mb, teapot.env scene
	Tomcatv	256 points, 5 iterations
	Unstructured	mesh.2k, 5 time steps
	Waternsq	512 molecules, 4 time steps
	Watersp	512 molecules, 4 time steps
PARSEC	**Blackscholes**	simmedium
	fluidanimate	simmedium
	Swaptions	simmedium

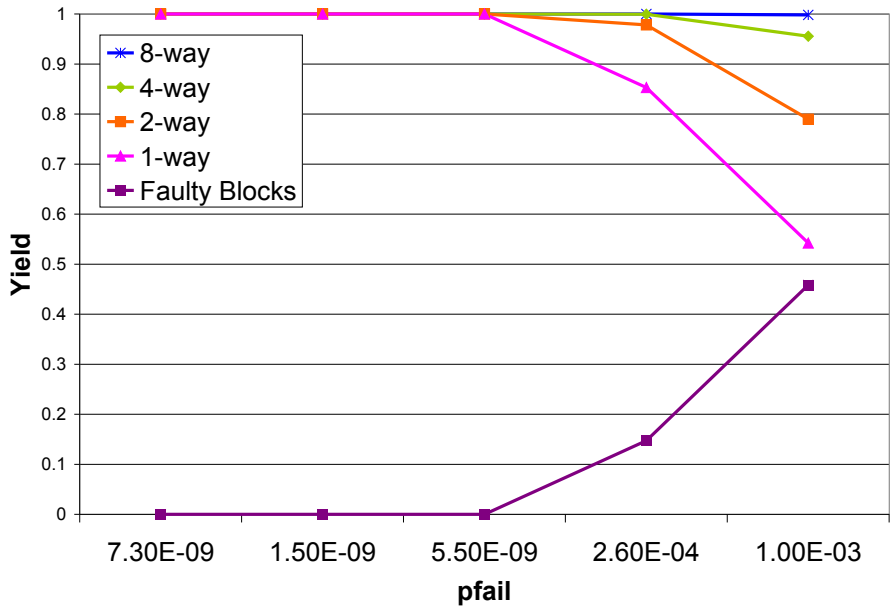
TABLE 6.3: Evaluated applications and input sizes.

6.5.1 Yield Analysis

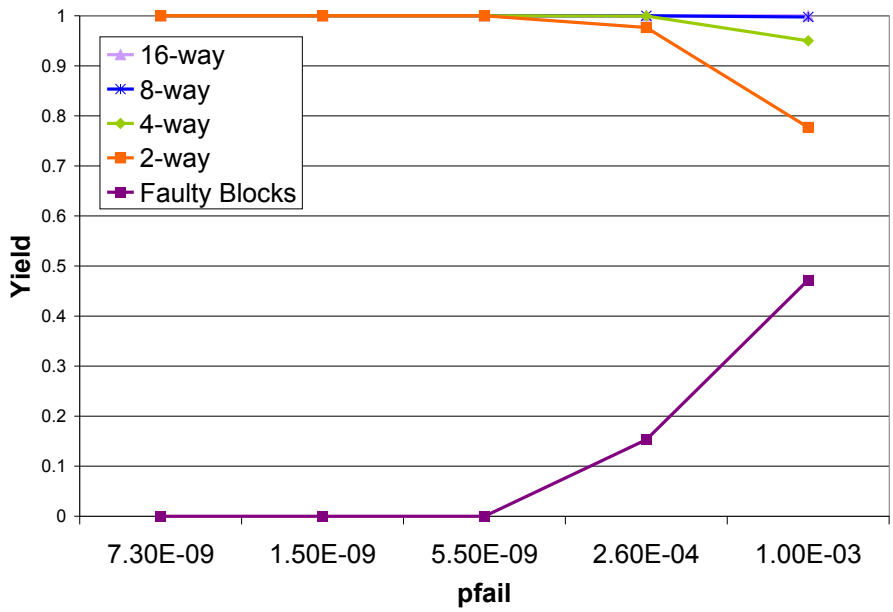
One important tool to understand the behaviour of a cache prone to permanent faults is yield analysis. The yield is a statistical metric referred to as the proportion of valid caches (or other components) from a population. In our case, we argue that a cache is invalid when, at least, all the blocks forming a set are disabled because of permanent faults.

In Figure 6.1 we can see a yield analysis for 32KB and 512KB caches for the different technology p_{fails} as depicted in Table 6.1. The study shows that, for a given p_{fail}, the yield increases with the number of ways. This is reasonable since the probability of having a disabled set (all blocks are faulty) is lower if the number of ways is higher. Despite the fact that for low p_{fails} the yield remains virtually imperturbable, we start to notice a dangerous trend for a p_{fail} of 2.6e-04 (expected for a 12nm technology) and more in a 32KB cache (a typical L1 configuration). Additionally, we can see in the graphs the percentage of faulty blocks in relation to the p_{fail}. As expected, this percentage clearly grows with the increase of the

p_{fail}. However, this estimation is not useful to determine the impact over the cache since it depends on the location of the faults. Finally, it is worthy to mention than Yield is barely affected if associativity is increased over 8 ways.



(a) 32KB cache



(b) 512KB cache

FIGURE 6.1: Yield versus percentage of faulty blocks for different p_{fails} in 32KB and 512KB caches.

6.5.2 Methodology Validation

Before proceeding to extract the EMR using our methodology, we first need to assure that it is valid. For that purpose, we compare our analytical model against the use of randomly generated maps of faults.

In Figure 6.2 we can see the probability distribution of the number of faulty blocks for different p_{fails} (we omitted 6.1e-13 because it offers around 0 and 1 faulty block) in a 32KB 8-way associative cache with 615 bits per block[3]. Results show the estimated faulty blocks provided by our model (analytical) and by different numbers of faulty maps (from 100 to 10 millions). As it is observed, few faulty maps are not able to catch the exact behaviour of the analytical model. However, we can see that when the number of maps is increased (10K maps or more) the number of faulty blocks per cache is accurately obtained. Nonetheless, this study cannot conclude how adequate approximation random maps are to estimate the expected misses of a cache since they directly depend on the location of faults in the different sets of the cache.

6.5.3 EMR and SD_MR for Sequential Benchmarks

In this Section we show the estimated EMR for several sequential benchmarks in a 8-way 32KB L1 cache which is subjected to permanent faults with different p_{fails}.

Figure 6.3(a) shows the relative EMR for different applications depending on the p_{fail} of the architecture as provided by our model. EMR relative increases are highly application-dependant. For instance, we can see that *twolf* starts to degrade cache performance in a 14% with respect to a base case without faults when the p_{fail} is 5.5e-05, while others like *gap* are barely affected. At the same time, we notice that SD_MR also grows in relation to the p_{fail} and that this growth is application-dependant as well. This trend is more clearly seen with a p_{fail} of 2.6e-04 which corresponds to a 12nm technology. In Figure 6.3(b) we can see the EMR and SD_MR for the same benchmarks in a 8-way 512KB L2 cache[4]. The

[3]We consider L1 cache blocks comprised of: 64 bytes for data and 8 bytes for its ECC, 28 bits for the tag and 1 byte for its ECC, and 3 control bits for valid, disable and dirty states.

[4]L2 blocks are comprised of 645 bits, which include the directory information and its ECC.

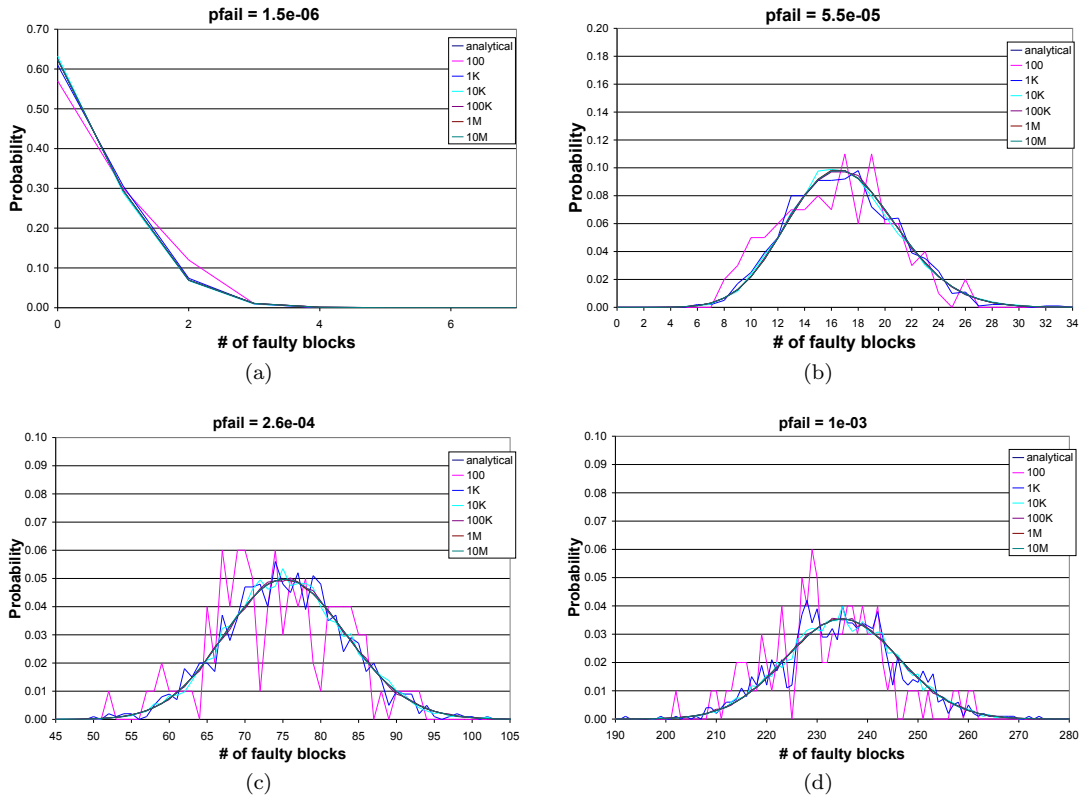
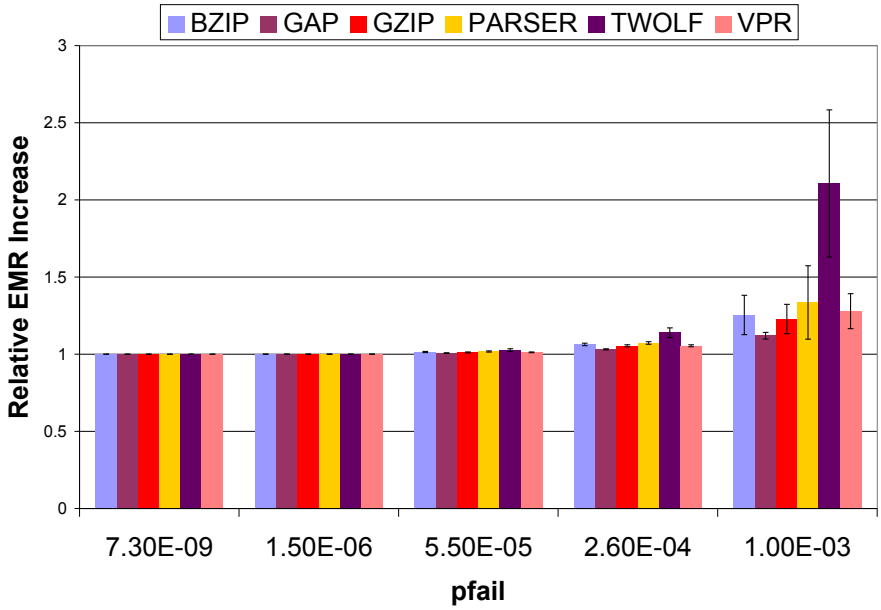


FIGURE 6.2: Probability distribution of the number of faulty blocks per cache obtained analytically and by randomly generated maps.

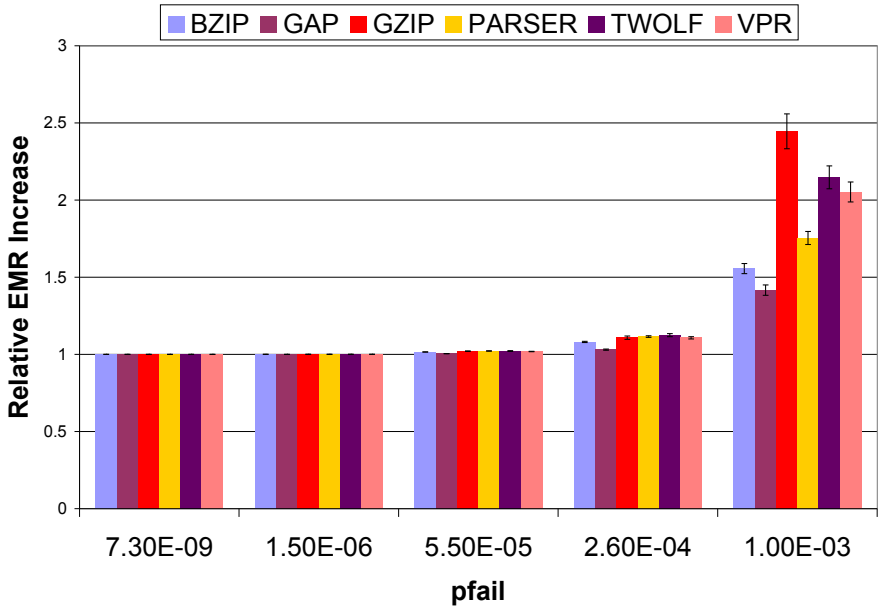
first thing we notice is that EMR relative increases reflected for the L1 are not the same for the L2. This clearly shows that the EMR is dependant of the pattern of accesses of every application to every cache. Finally, we can see that the EMR relative increase is more moderate in L2. This is explained because of the effect of associativity, which is able to absorb most of the penalty because of cache capacity reduction.

In Section 6.5.2 we showed how fault maps can approximate the number of faulty blocks per cache. Nonetheless, it is still unknown how adequate they are to approximate to the EMR provided by our model. The answer can be found in Figure 6.4, which presents the EMR for different applications in a 8-way 32KB L1 cache. With 10 faulty maps, applications such as *bzip* or *twolf*, are not able to capture the behaviour of the model for the highest p_{fails}, while differences are barely noticeable with the smallest p_{fails}. This is due to the fact that the number

of affected blocks is very small and the effect over the total number of misses is negligible. However, with a higher number of maps the EMR and SD_MR converge to the one provided by the analytical model. In general, we can state that 1,000 maps are able to produce acceptable approximated values for both EMR and SD_MR.



(a) 8-way 32KB L1 cache



(b) 8-way 512KB L2 cache

FIGURE 6.3: EMR and SD_MR relative increase for sequential applications.

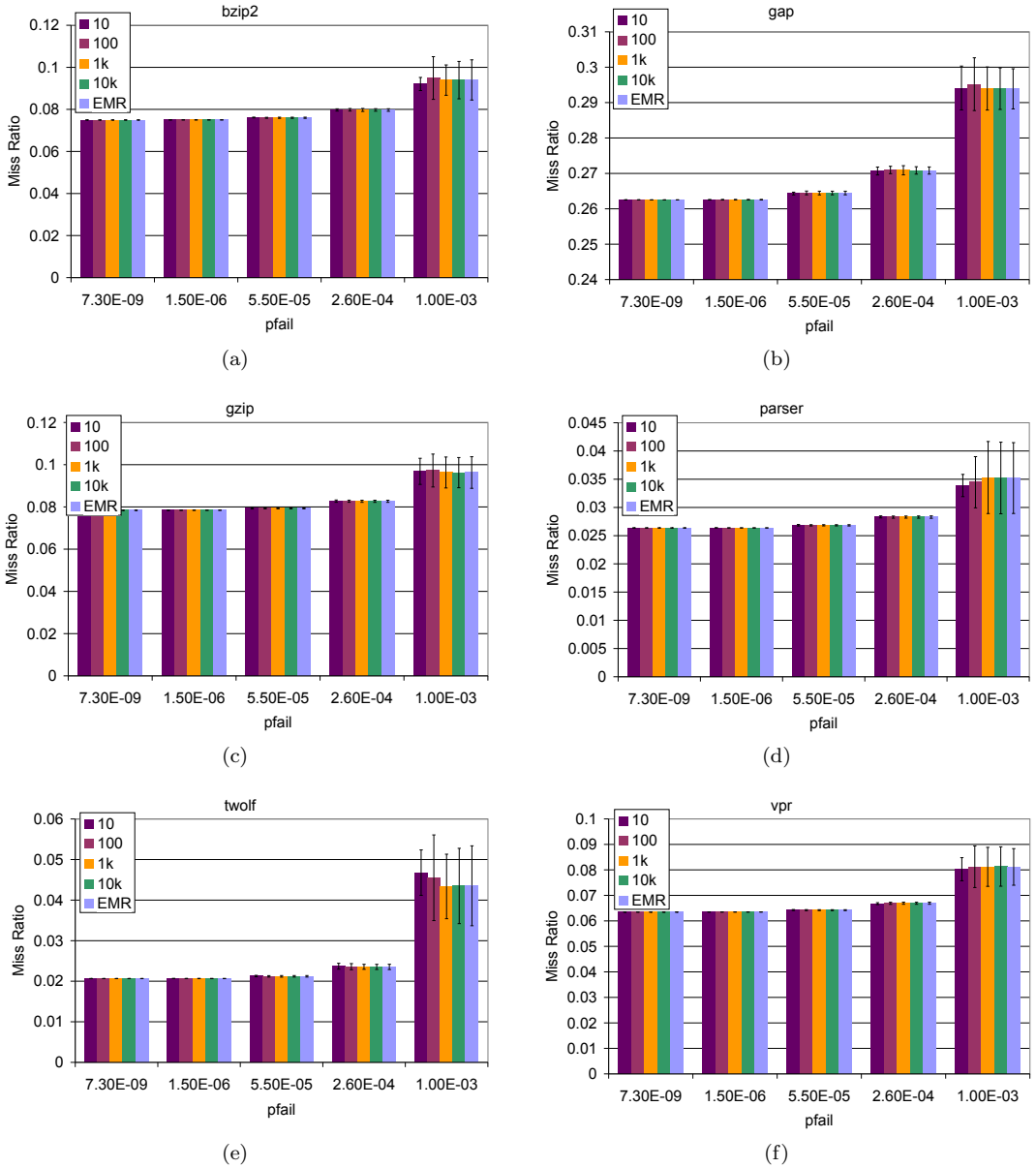


FIGURE 6.4: EMR and SD_MR for different applications in a 8-way associative 32KB L1 cache with different p_{fails}.

Surprisingly, this study shows that even a low number of faulty maps are enough to approximate the EMR. The reason for this is the access homogeneity to the different sets of the cache. In other words, for real-world applications as the ones we have evaluated, there are no particular sets that are clearly more accessed than others along the overall execution of the benchmark. This makes the EMR virtually independent of the allocation of the faults and that is the reason why fault maps are able to provide such satisfactory estimations.

In order to prove the cache access homogeneity we refer to, we have performed a study of the correlation of accesses between all the sets in our cache. In this case we have used the Pearson correlation coefficient. Given X and Y, two different sets of our cache with different accesses per way

$$X = x_1 ... x_n, Y = y_1 ... y_n \tag{6.12}$$

The Pearson correlation coefficient is calculated as:

$$r_{xy} = \frac{\sum_{i=1}^{N}(x_i - \bar{x}) \cdot (y_i - \bar{y})}{\sqrt{\sum_{i=1}^{N}(x_i - \bar{x})^2} \cdot \sqrt{\sum_{i=1}^{N}(y_i - \bar{y})^2}} \tag{6.13}$$

Benchmark	Mean Pearson Coeff.	DEV
bzip	.997	.007
gap	.995	.009
gzip	.998	.005
parser	.998	.002
twolf	.995	.007
vpr	.998	.004

TABLE 6.4: Average for the Pearson Coefficient Matrix for every benchmark.

When r_{xy} is close to 0 it means that there is no correlation between variables, whereas when it is close to 1 there exist a relation between them. We have calculated the matrix of correlations for the number of accesses to a 8-way 32KB L1

cache for the SPEC benchmarks. Table 6.4 reflects the average value for the Pearson coefficients as well as its standard deviation. As we can see, all coefficients are very close to 1, which means that the accesses among sets are highly correlated.

But still we can study the significance of the Pearson correlation coefficient to asses the correlation among the variables is true as was not produced arbitrarily. For that, we can calculate Student's t as:

$$t = \frac{r_{xy}}{\sqrt{\frac{1-r_{xy}^2}{N-2}}} \tag{6.14}$$

in which N is the number of values for the variables. In our example, for a 8-way cache, we have that $N = 8+1$ taking into account the number of cold misses. For the worst correlation exhibited by the studied benchmarks, which is *twolf* with .995, we have that $t = 10.63$. By means of a calculator for the t distribution, for a degree of liberty $N = 7$, we can determine that our sets are correlated with a maximum risk of $1e-5$ corroborating our hypothesis.

The lesson we extract from this study is that because of the high correlation, the amount of random fault maps necessary to approximate our model is low. In the case that data accesses among sets were not so correlated, a few fault maps would not be able to extract the behaviour of miss ratio in faulty caches.

6.5.4 EMR Probability Distribution for Sequential Applications

As explained in Section 6.3.3 we have developed a mechanism to calculate a probability distribution for the expected values for the EMR, which we call PD_MR. To afford this, we follow a constructive approach. On the one hand, there is the probability of having n faulty blocks in our cache, which we already know. On the other hand, we calculate for every possible number of faults in the cache (from 0 to the total number of blocks in the cache) which one is the most likely p_{fail} to obtain that amount of faults. Then, we can calculate the EMR for that p_{fail}. This way, we have all the necessary elements to plot the PD_MR distribution.

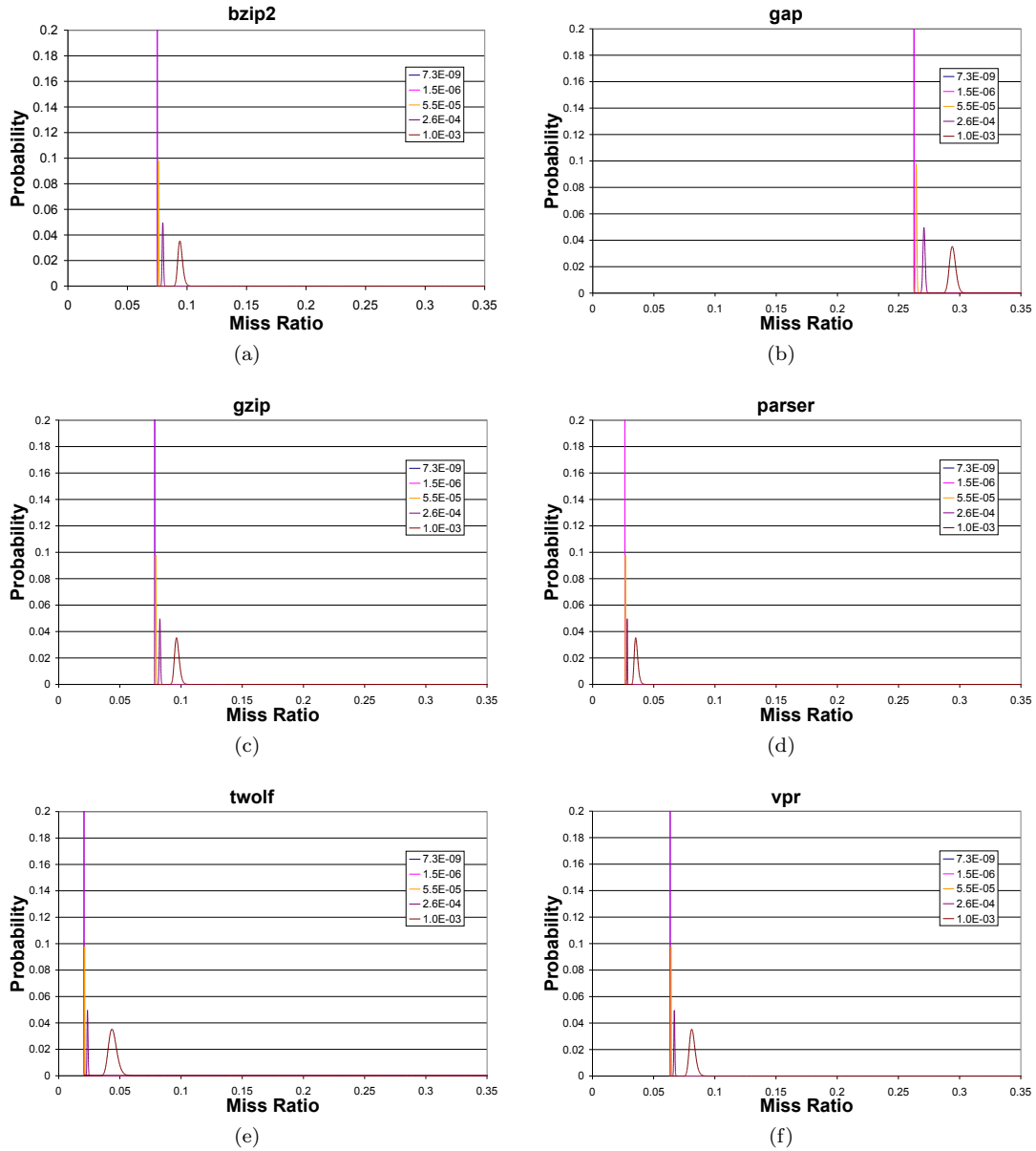


FIGURE 6.5: PD_MR for different applications and p_{fails} in a 8-way associative 32KB L1 cache.

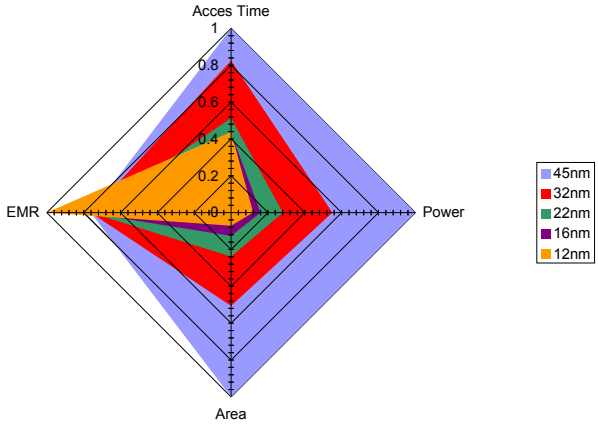
In Figure 6.5 we can see the PD_MR distributions for all the SPEC benchmarks studied for a 8-way associative 32KB L1 cache (the same configuration used in the previous Section). As it can be observed, the calculated distribution matches perfectly with the EMR calculated by our first model as can be seen in Figure 6.4. This shows the accuracy of our probability distribution model.

However, the probability distribution model provides a more valuable information: we can clearly see how bad the effect of permanent faults could be. This PD_MR model could be used by chip manufacturers to analytically see what the percentage of chips which should be discarded resulting from of a too damaged cache is, for example.

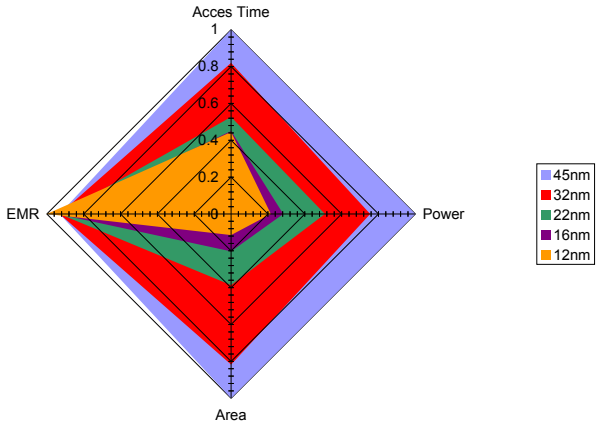
6.5.5 Cache Performance Trade-Offs for Sequential Applications

Figure 6.6 plots the EMR for a 32KB L1 cache with different associativity and scaling technologies in comparison to its access time, area and power dissipation, as extracted from the cache simulator CACTI [104]. In general, a lower scaling technology effectively reduces the access time, power and area requirements for a given cache whereas it increases the EMR, due to the higher p_{fail}. This effect is more clearly seen in a 2-way cache like in Figure 6.6(a), in which a technology of 12nm increases the EMR with respect to 16nm in 22%, whereas the latter increases the required area in 43%, the power dissipation in 24%, at the same time it achieves a similar access time. In case that the primary goal is to reduce the performance impact on the cache, 16nm offers a better EMR/access-time trade-off than 12nm.

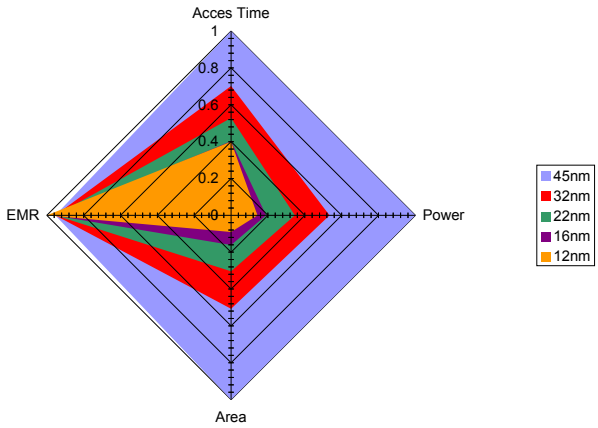
However, with the use of a higher associativity the EMR is decreased. E.g., as depicted in Figures 6.6(b) and 6.6(c), for 4-way and 8-way associative caches, the EMR difference between 16nm and 12nm technologies is 5.6% and 3.5%, whereas the difference in the access time is 1.2% and 0.1% respectively. To sum up, this EMR/access-time trade-off study suggests that the use of the smallest scaling technology is advisable only with a relatively high associativity which is able to hide the increased p_{fail}.



(a) 2-way L1 32KB cache.



(b) 4-way L1 32KB cache.



(c) 8-way L1 32KB cache.

FIGURE 6.6: Trade-off among different scaling technologies and cache configurations.

6.5.6 EMR Impact of Block Disabling and Word Disabling

In this Section, we compare the performance implications of word-disabling (*wdis*), a technique proposed by Wilkerson *et al.* [111] to enable correct cache operation in the presence of faults.

The *wdis* technique tracks faulty data cells at word granularity by means of fault masks. These masks are kept at every line's tag and contain as many bits as words, each one indicating whether the corresponding word is disabled or not. When the *wdis* technique is deactivated the fault mask is ignored. However, when it is active, every pair of consecutive blocks in a set are combined to form one logical block. The effect of this mechanism is that both the capacity and associativity of the cache are reduced by half.

In order to obtain a logical block in aligned form, *wdis* introduces a shift-multiplexer network which is controlled by each block's fault mask, which has the effect of discarding the defective words. This way, *wdis* tolerates up to $n/2$ faulty words in a logical block with n words. Unfortunately, the alignment network increases the access latency of the cache. Specifically, for 8-word blocks, *wdis* requires a line to pass through 4 different multiplexers (to discard up to 4 faulty words), something which increases the latency of the cache in one cycle, as reported in [111].

In Figure 6.7 we can see the EMR of block-disabling and word-disabling provided by our model, for a 32KB L1 cache with a different number of ways for the SPEC applications. The first thing which is worth of notice is that word-disabling tolerates all introduced faults without additional performance cost (despite of the reduced cache capacity and associativity). Therefore, the EMR for the different p_{fails} remains constant.

As we can see in Figure 6.7, up to the p_{fail} of 5.5e-05, which corresponds to a technology of 16nm (according to Table 6.1), block-disabling obtains lower EMR than word-disabling for each configuration. However, with a p_{fail} of 2.6e-04 word-disabling results in a lower EMR for 2-way caches. Furthermore, the deviation starts to grow noticeably in block-disabling, whereas in word-disabling remains

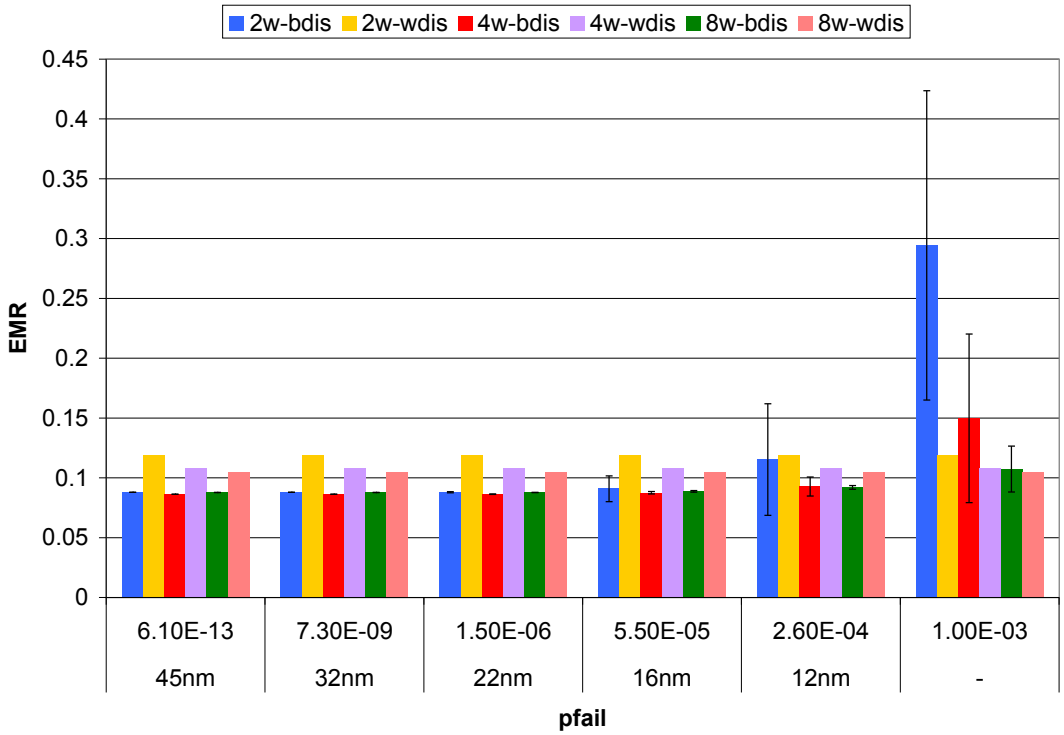


FIGURE 6.7: EMR for block-disabling and *wdis* in a 32KB L1 Cache.

constant, as explained before. Finally, with a p_{fail} of 1.0e-03, word-disabling obtains lower EMR than block-disabling for each configuration.

In conclusion, we can remark that, for low p_{fails} block-disabling is an attractive design in terms of EMR impact, whereas for high p_{fails} word-disabling behaves better. Nonetheless, for a medium to high number of ways, the performance of both techniques is quite similar. In any case, we have to remind that, while the block-disabling technique does not incur in any latency overhead, word-disabling increases latency and needs to include additional hardware for the shift-multiplexer network, which could make block-disabling more suitable to deploy in real environments.

6.5.7 EMR and SD_MR for Shared Caches in Parallel Benchmarks

Figure 6.8 shows the EMR for several parallel benchmarks in an 8-way 512KB L2 cache. We have omitted some of them for the sake of visibility and we mereley show the most characteristic ones. As it is observed once again, randomly generated maps are able to capture the trend of our analytical model.

In this case, the homogeneity of accesses to the L2 is even higher than in the case of the L1 cache, so the behaviour of the faulty maps is expected. The reason is twofold: firstly, because the L2 accesses were already filtered by the L1 cache. Frequently accessed memory locations tend to remain in the L1 (due to locality). Therefore the existence of *hot blocks* in the L2 is even more unlikely than in the L1. Secondly, due to the high associativity, which limits the impact of faults increasing the accuracy of fault maps.

6.5.8 Implication of the Number of Threads in EMR and SD_MR

Another use-case example for our analytical model is to study how the EMR is affected when varying the number of threads per application. In this study, each thread runs in a different core of a CMP. This way, both the number of cores and the number of private L1 caches increase at the same time as the number of threads. The size of the shared L2 cache remains constant.

Figure 6.9(a) depicts the EMR and SD_MR for all the studied parallel benchmarks. The general trend is that the EMR increases with the number of threads. This is explained by the fact that the number of L2 accesses decreases with the number of threads due to the increased L1 capacity, as depicted in Figure 6.9(b). Thus, since the input size of the different benchmarks remains constant, the misses per access ratio (EMR) increase with the number of threads. As derived from Figure 6.9(a), this trend remains constant for all the studied p_{fails} and the relative

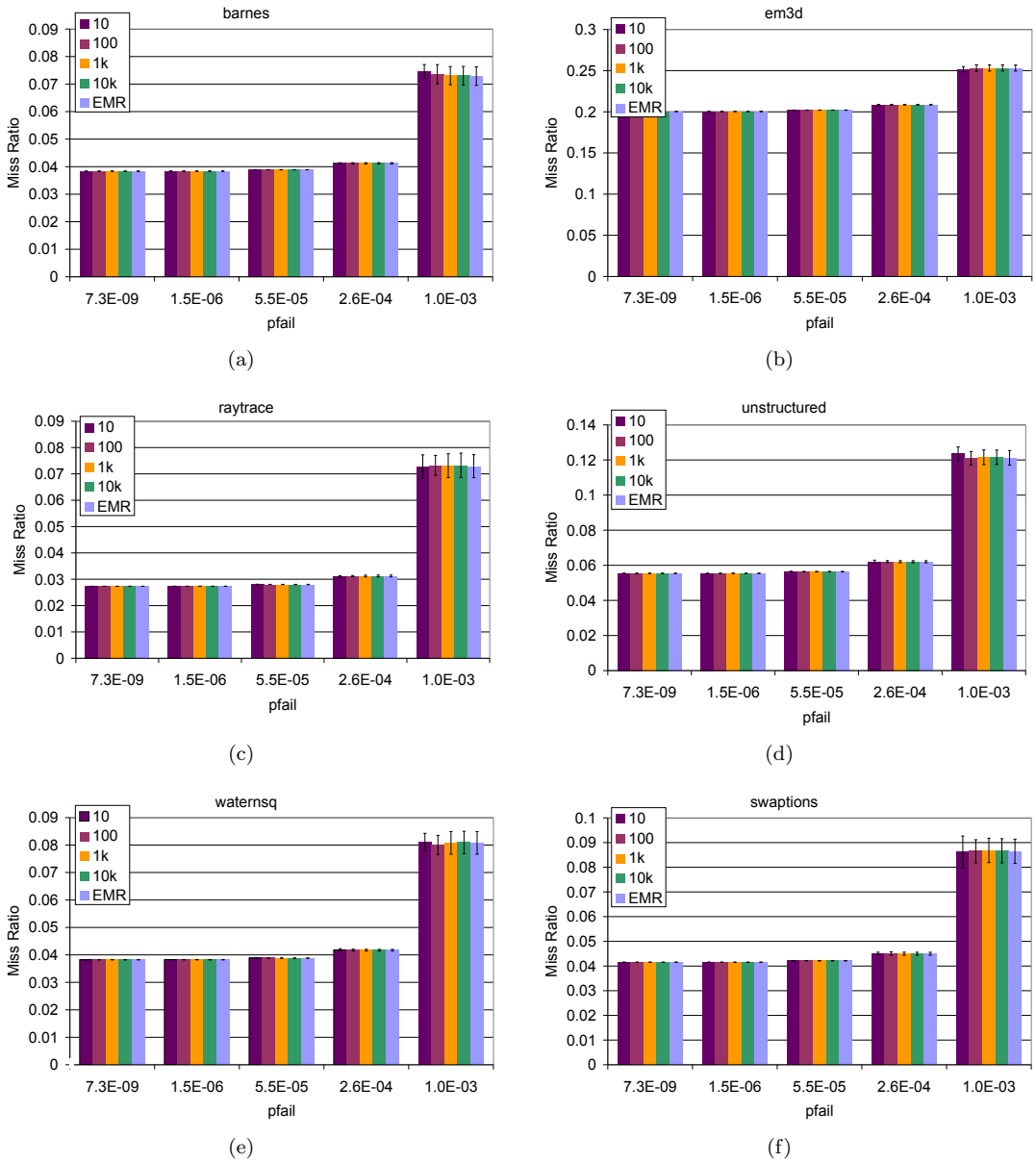
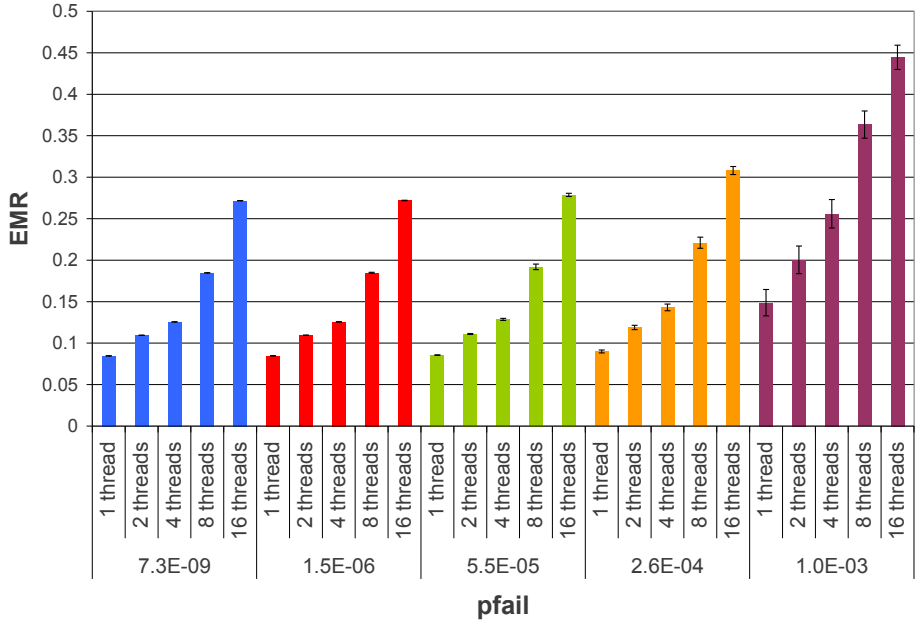
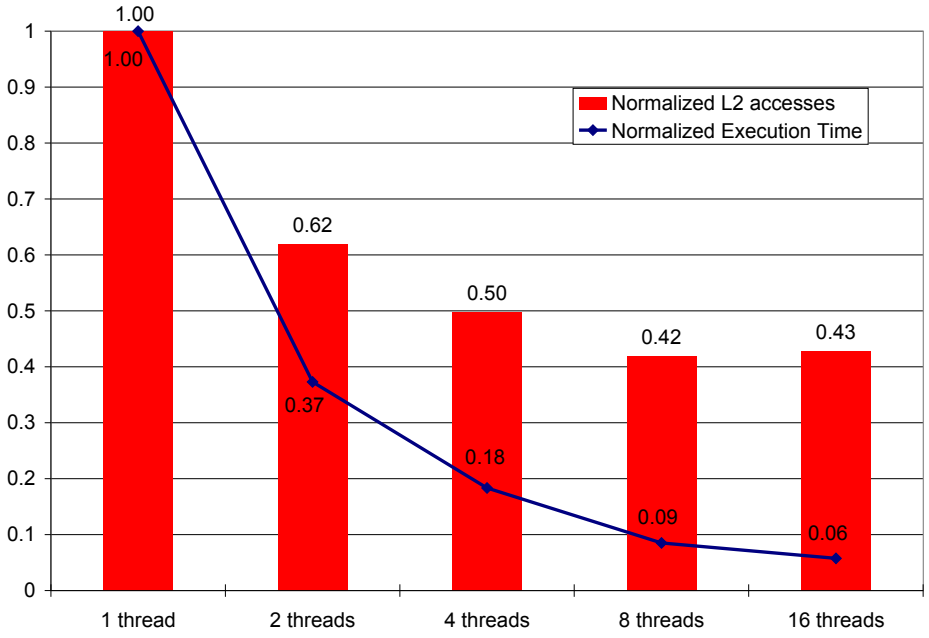


FIGURE 6.8: EMR and SD_MR for different applications in a 8-way 512KB L2 cache with different p_{fails}.

EMR increases 1.3 for 2 threads, 1.5-1.7 for 4 threads, 2.1-2.4 for 8 threads and 3-3.4 for 16 threads, with respect to the EMR for 1 thread. Therefore, we can conclude that, in the presence of faults, applications will experience a higher performance impact as the number of threads increases.



(a) EMR and SD_MR.



(b) Normalized execution time and L2 accesses count in a fault-free environment.

FIGURE 6.9: Results for a 8-way 512KB L2 cache for parallel applications.

6.6 Concluding Remarks

In the past years, technology scaling has enabled continuous miniaturization of circuits and wires offering designers the opportunity to place more functionality per unit area. Furthermore, the increased device density has allowed the integration of large caches and many cores into the same chip. However, this trend has a major drawback in the growth of the failure ratio in every new scale generation. In particular, SRAM cells are very susceptible to the use of decreasing voltages, higher frequencies and temperature, and other events such as power supply noise, signal cross-talking and process variation which eventually result in severely affecting the reliability of caches.

There is a body of literature on dealing with the occurrence of permanent faults in caches. However, all these previous studies rely on the use of random fault maps, typically a small subset of the actual number of possible mappings. Therefore, it remains unclear whether the results presented in these studies are representative and/or accurate enough.

In this Chapter, it is proposed an analytical model to determine the Expected Miss Ratio (EMR) for a given application whenever it is executed in a cache with a random probability of cell failure (p_{fail}). This analytical model allows designers to perceive the real impact of faults in caches, including the standard deviation of the EMR, without the need of executing any iterative analysis with random maps. We have also presented a second analytical model which provides the probability distribution for the EMR which represents even more valuable information for designers about the deployment of faulty cache units for a given process technology.

In the evaluation we show how, for the benchmarks and configurations used, the random fault map methodology provides high accuracy when using 1000 and 100 maps for the L1 and L2 caches, respectively. This is due to the high data access homogeneity to the different sets of a cache which makes the EMR virtually independent of the allocation of faults, and resulting in a not very high number of random faulty maps able to capture the mean and standard deviation for the EMR.

Additionally, we have studied the EMR, access time, power, and area trade-offs of different cache configurations. The main conclusion extracted is that the use of very low scaling technologies (which results in increasing p_{fails}) is not advisable for low associative caches, e.g. 2-way caches, because of the the impact on EMR.

Finally, we have compared block-disabling with word-disabling. Results show that, for higher p_{fails}, word-disabling is more adequate due to its reduced EMR. Nonetheless, for highly associative caches the performance of block-disabling approaches the word-disabling one, whereas the former does not increase cache latency.

Chapter 7

Conclusions and Future Ways

7.1 Conclusions

Over the last decades we have benefited from the scaling technology to improve the performance of computer systems. However, increasing device-level variation, unpredictability and failures will surely challenge this performance-cost trade-off in the next 10-20 years. According to Moore's law, industry has grown exponentially in every new scale generation. But this has taken its toll: increased variability and fault rates. The problem is that, if an increasing part of the new hardware must be devoted to implement fault tolerant techniques, the imposed performance goals may not be satisfied and the growth of computer industry may no longer be sustained.

This multi-variable trade-off has two major implications. First, the increasing fault rate and variation, which is avoiding traditional solutions to provide with enough reliability and increasing the hardware costs. And second, the effect of power constraints, which are motivating aggressive mechanisms to reduce supply voltage below safe margins at the cost of a system's reliability decrease.

In this upcoming scenario, some previous mechanisms address the fault tolerance challenge at circuit level. However, this requires a complete redesign of circuits increasing complexity design and degrading performance due to the collateral

effects over critical paths. Thus, in this Thesis we have focused on architectural-level measures which avoid most of the aforementioned drawbacks. Traditionally, at this level, fault tolerance has been achieved by means of some sort of redundancy. Specifically, there is a large body of literature which focuses on Redundant Multi Threading (RMT), in which instructions are executed twice and outputs are compared in order to detect faults. Whereas previous works focused on monocore environments and sequential benchmarks, the architectural support for shared-memory applications in parallel and scalable environments has remained under-explored.

The major problems which we have identified in previous RMT approaches are memory consistency and performance related. In order to solve these issues, in this Thesis we address the performance-cost challenge by means of two different mechanisms which are specifically deployed in scalable direct-network environments which will constitute future multi-core architectures.

We have analyzed previous proposals in Chapter 4. We found that, without proper support, CRT(R) approaches could lead to memory consistency violations because of their cache update policy. In order to avoid the propagation of errors, CRT(R) only updates memory after a successful verification of data values. But this represents a risk in shared-memory applications because, if not properly treated, the access to shared-memory regions in isolation must be compromised. Additionally, we studied the impact of moving Dynamic Core Coupling (DCC) into a direct-network environment. The result is that the indirection caused by this kind of interconnection network dramatically affects DCC's performance due to the extra communication overhead. To solve these issues, we proposed an alternative RMT approach based on CRT(R) called REPAS, in which redundant threads run in a 2-way SMT cores in a tiled-CMP. As a way to eliminate memory consistency related problems, we proposed to update memory speculatively (before redundant verification), although buffering memory values to avoid fault propagation.

We showed through our simulated experiments that our REPAS approach is able to outperform CRT(R) and DCC (which uses twice the amount of hardware). REPAS reduces the total overall execution overhead with respect to a base case

with no fault tolerant mechanisms down to 25% in a fault-free environment. Additionally, we showed that the performance of REPAS is barely affected in a faulty environment, even if extremely high fault rates are employed. The two major drawbacks of REPAS are, first, the performance degradation resulting from the resource sharing between redundant pair threads in SMT cores and, second, the added complexity derived from the implementation of different buffers to communicate redundant threads.

In Chapter 5 we address these shortcomings based on an already proposed Hardware Transactional Memory (HTM) architecture, LogTM-SE. In our presented mechanism, LBRA, redundant threads communicate through a virtual memory log placed in cache to provide both input replication and recovery after the detection of a fault. LBRA avoids increased hardware complexity by using the already present hardware in LogTM-SE. Furthermore, LBRA allows the programmer to explicitly enable and disable specific portions of the program to be protected by means of redundancy.

In our initial LBRA approach, redundant threads are executed in the same dual-threaded SMT core, providing a low-resource overhead solution. The counterpart is the performance degradation inherent to SMT architectures. Our second approach avoids this pitfall by executing redundant threads in different (non-SMT) cores. In this environment, the major drawback is related to the inter-core communication. We solve this issue by means of a set of measures such as a log buffer, a prefetch strategy and slight modifications of specific coherence actions.

Our analysis showed that LBRA outperforms both REPAS and DCC in the same environment. Specifically, the execution time overhead of LBRA is 20% for the coupled approach and 7% for the decoupled one, with respect to a base case without fault tolerance mechanisms in a fault-free environment. At the same time, we show that LBRA performs better in a faulty environment than previous proposals, when injecting random faults even at a rate of 100 faults per million of cycles.

Miniaturization has a major drawback in the growth of the failure ratio in every new scale generation. Specifically, SRAM cells are remarkably susceptible

to the use of decreasing voltages and higher frequencies and temperatures, which decreases significantly the reliability of cache memories. While the use of ECC codes is commonly extended to solve this issue, in the presence of a large number of hard faults ECC is not an advisable solution because of its performance limitations. Instead, in Chapter 6 we study the impact of word/block disabling techniques, which avoid the use of faulty portions of the cache. For this, we present an analytical model to determine the implications of block-disabling due to random cell failure on cache miss rate behaviour. Contrarily to previous proposals based on the execution of a large number of fault maps, the presented analytical model provides an exact calculation for the expected miss ratio by a single pass over an application memory access trace, probability of cell failure and cache configuration.

7.2 Future Ways

The results presented in this Thesis open a number of interesting new research paths which we detail now:

- The performance overhead of REPAS is directly related to resource sharing, something inherent to the use of SMT architectures. However, it is our belief that smarter fetch policies could obtain a noticeable benefit. For instance, one idea is the design of a mechanism to detect the execution of critical path instructions. This way, affected threads could increase their priority, which would result in the improvement of overall performance. Another interesting approach would consist of using master thread results for critical path instructions as value predictions for the slave thread. This way we could obtain a noticeable performance speedup at the expense of decreasing fault coverage.

- Whereas we make use of an eager-eager (version management-conflict detection) policy in LBRA, as a future work we plan to study the behaviour of lazy-lazy and lazy-eager policies. When using a lazy version management, memory updates are buffered in a hardware structure which holds memory values during the execution of a transaction. The benefit for this approach

is twofold. First, memory remains unmodified until a successful verification, something which avoids the use of other mechanisms to bypass memory values between redundant threads. And second, in case of fault, the rollback recovery mechanism simply discards the buffered memory values. However, although this approach would be enough for the correct execution of single-threaded applications, the support for shared-memory applications needs additional mechanisms to avoid input incoherences, something which should be addressed by means of active conflict detection policies. Finally, we plan to extend the presented mechanisms to also work with transactional memory applications. For that, we need to add the support to distinguish between real transactions and pseudo-transactions and treat them accordingly, i.e. activating conflict detection mechanisms and aborting transactions when necessary.

- The presented analytical model focuses on random variations. However, several studies reveal that defects in ICs appear in clusters. This event impacts on both the yield and performance of affected devices. As future work, we plan to develop our model to take into account the behaviour of systematic variations which occur in clusters. Additionally, we would like to adapt the model to estimate the EMR when considering both L1 and L2 as faulty at the same time. We would also like to use our cache model to derive a performance model for the whole processor.

Bibliography

[1] A. Agarwal, B. Paul, S. Mukhopadhyay, and K. Roy. Process variation in embedded memories: failure analysis and variation aware architecture. *IEEE Journal of Solid-State Circuits*, 40(9):1804 – 1814, 2005.

[2] K. Agarwal and S. Nassif. Statistical analysis of sram cell stability. In *Proceedings of the 43rd annual Design Automation Conference*, pages 57–62. ACM, 2006.

[3] AMD. Bios and kernel developer's guide for amd athlon^{TM}64 and amd opteron^{TM}processors. *Publication #26094, Revision 3.14, April 2004.*

[4] T. M. Austin. Diva: a reliable substrate for deep submicron microarchitecture design. In *Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*, pages 196–207. IEEE Computer Society, 1999.

[5] J. Bartlett, J. Gray, and B. Horst. Fault tolerance in tandem computer systems. In *The Evolution of Fault-Tolerant Systems*, pages 55–76. Springer-Verlag, 1987.

[6] R. Baumman. Soft errors in advanced computer systems. *IEEE Design and Test of Computers*, 22:258–266, May 2005.

[7] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The parsec benchmark suite: characterization and architectural implications. In *Proceedings of the 17th*

International Conference on Parallel Architectures and Compilation Techniques, pages 72–81. ACM, 2008.

[8] C. Blundell, M. M. Martin, and T. F. Wenisch. Invisifence: performance-transparent memory ordering in conventional multiprocessors. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, pages 233–244. ACM, 2009.

[9] S. Borkar. Design challenges of technology scaling. *IEEE Micro*, 19(4):23 –29, 1999.

[10] S. Borkar, T. Karnik, S. Narendra, J. Tschanz, A. Keshavarzi, and V. De. Parameter variations and impact on circuits and microarchitecture. In *Proceedings of the 40th annual Design Automation Conference*, pages 338–342. ACM, 2003.

[11] K. Bowman, J. Tschanz, C. Wilkerson, S.-L. Lu, T. Karnik, V. De, and S. Borkar. Circuit techniques for dynamic variation tolerance. In *Proceedings of the 46th Annual Design Automation Conference*, pages 4–7. ACM, 2009.

[12] J. Carretero, P. Chaparro, X. Vera, J. Abella, and A. Gonzalez. Implementing end-to-end register data-flow continuous self-test. *IEEE Transactions on Computers*, 99(PrePrints), 2010.

[13] J. Carretero, X. Vera, P. Chaparro, and J. Abella. On-line failure detection in memory order buffers. In *IEEE International Test Conference*, pages 1 –10, 2008.

[14] L. M. Censier and P. Feautrier. A new solution to coherence problems in multicache systems. *IEEE Transactions on Computers*, 27:1112–1118, 1978.

[15] L. Ceze, J. Tuck, P. Montesinos, and J. Torrellas. Bulksc: bulk enforcement of sequential consistency. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, pages 278–289. ACM, 2007.

[16] J. Clement. Electromigration modeling for integrated circuit interconnect reliability analysis. *IEEE Transactions on Device and Materials Reliability*, 1(1):33 –42, Mar. 2001.

[17] C. Constantinescu. Trends and challenges in vlsi circuit reliability. *IEEE Micro*, 23(4):14 – 19, 2003.

[18] D. Culler, J. P. Singh, and A. Gupta. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kauffman, 1998.

[19] E. N. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys*, 34:375–408, 2002.

[20] E. N. Elnozahy and W. Zwaenepoel. Manetho: transparent roll back-recovery with low overhead, limited rollback, and fast output commit. *IEEE Transactions on Computers*, 41(5):526 –531, 1992.

[21] R. Fernández. *Fault-tolerant Cache Coherence Protocols for CMPs*. PhD. Thesis, 2009.

[22] D. J. Frank. Power-constrained CMOS scaling limits. *IBM Journal of Research and Development*, 46(2/3):235–244, 2002.

[23] A. Gefflaut, M. Banatre, A. Kermarrec, and C. Morin. Coma: An opportunity for building fault-tolerant scalable shared memory multiprocessors. In *23rd Annual International Symposium on Computer Architecture*, pages 56–65, 1996.

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

[25] M. Gomaa, C. Scarbrough, T. N. Vijaykumar, and I. Pomeranz. Transient-fault recovery for chip multiprocessors. In *Proceedings of the 30th International Symposium on Computer Architecture*, pages 98 – 109, 2003.

[26] M. A. Gomaa and T. N. Vijaykumar. Opportunistic transient-fault detection. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 172–183. IEEE Computer Society, 2005.

[27] A. González, S. Mahlke, S. Mukherjee, R. Sendag, D. Chiou, and J. J. Yi. Reliability: Fallacy or reality? *IEEE Micro*, 27(6):36–45, 2007.

[28] J. Henning. Spec cpu2000: measuring cpu performance in the new millennium. *Computer*, 33(7):28 –35, July 2000.

[29] M. D. Hill and A. J. Smith. Evaluating associativity in cpu caches. *IEEE Transactions on Computers*, 38:1612–1630, December 1989.

[30] D. Hunt and P. Marinos. A general purpose cache-aided rollback error recovery (carer) technique. In *In Proceedings of the 17th International Symposium on Fault-Tolerant Computing Systems*, pages 170 –175, 1987.

[31] T. Ishihara and F. Fallah. A cache-defect-aware code placement algorithm for improving the performance of processors. In *IEEE/ACM International Conference on Computer-Aided Design*, pages 995 – 1001, Nov 2005.

[32] T.-H. Kim, J. Liu, J. Keane, and C. Kim. A 0.2 v, 480 kb subthreshold sram with 1 k cells per bitline for ultra-low-voltage computing. *IEEE Journal of Solid-State Circuits*, 43(2):518 –529, 2008.

[33] S. Krumbein. Metallic electromigration phenomena. *IEEE Transactions on Components, Hybrids, and Manufacturing Technology*, 11(1):5 –15, 1988.

[34] R. Kumar, V. Zyuban, and D. M. Tullsen. Interconnections in multi-core architectures: Understanding mechanisms, overheads and scaling. In *Proceedings of the 32th International Symposium on Computer Architecture*. ACM, 2005.

[35] S. Kumar and A. Aggarwal. Speculative instruction validation for performance-reliability trade-off. In *Proceedings of the IEEE 14th International Symposium on High Performance Computer Architecture*, pages 405 – 414, 2008.

[36] N. Ladas, Y. Sazeides, and V. Desmet. Performance-effective operation below vcc-min. In *IEEE International Symposium on Performance Analysis of Systems Software*, pages 223 –234, 2010.

[37] C. LaFrieda, E. Ipek, J. F. Martinez, and R. Manohar. Utilizing dynamically coupled cores to form a resilient chip multiprocessor. In *Proceedings of the*

37th International Conference on Dependable Systems and Networks, pages 317–326, 2007.

[38] G. G. Langdon and C. K. Tang. Concurrent error detection for group look-ahead binary adders. *IBM Journal of Research and Development*, 14:563–573, 1970.

[39] H. Q. Le, W. J. Starke, J. S. Fields, F. P. O'Connell, D. Q. Nguyen, B. J. Ronchetti, W. M. Sauer, E. M. Schwarz, and M. T. Vaden. Ibm power6 microarchitecture. *IBM Journal of Research and Development*, 51(6):639 –662, 2007.

[40] H. Lee, S. Cho, and B. Childers. Exploring the interplay of yield, area, and performance in processor caches. In *25th International Conference on Computer Design*, pages 216 –223, 2007.

[41] H. Lee, S. Cho, and B. Childers. Performance of graceful degradation for cache faults. In *IEEE Computer Society Annual Symposium on VLSI*, pages 409 –415, 2007.

[42] M.-L. Li, P. Ramachandran, S. K. Sahoo, S. V. Adve, V. S. Adve, and Y. Zhou. Understanding the propagation of hard errors to software and implications for resilient system design. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 265–276. ACM, 2008.

[43] M.-L. Li, R. Sasanka, S. V. Adve, Y. kuang Chen, and E. Debes. The alpbench benchmark suite for complex multimedia applications. In *Proceedings of the IEEE International Symposium on Workload Characterization*, pages 34–45, 2005.

[44] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, and D. M. T. andNorman P. Jouppi. Mcpat: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In *42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 469–480, 2009.

[45] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, B. Werner, and B. Werner. Simics: A full system simulation platform. *Computer*, 35(2):50–58, 2002.

[46] A. Maheshwari, W. Burleson, and R. Tessier. Trading off transient fault tolerance and power consumption in deep submicron (dsm) vlsi circuits. *IEEE Transactions onVery Large Scale Integration Systems*, 12(3):299 – 311, 2004.

[47] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. Multifacet's general execution-driven multiprocessor simulator (gems) toolset. *SIGARCH Computer Architecture News*, 33(4), 2005.

[48] J. F. Martínez, J. Renau, M. C. Huang, M. Prvulovic, and J. Torrellas. Cherry: Checkpointed early resource recycling in out-of-order microprocessors. In *Proceedings of the 35th International Symposium on Microarchitecture*, pages 3 – 14, 2002.

[49] R. Mastipuram and E. C. Wee. Soft error's impact on system reliability. *Electronics Design, Strategy, News (EDN)*, pages 69–74, September 2004.

[50] Y. Masubuchi, S. Hoshina, T. Shimada, B. Hirayama, and N. Kato. Fault recovery mechanism for multiprocessor servers. In *27th Annual International Symposium on Fault-Tolerant Computing*, pages 184 –193. IEEE Computer Society, 1997.

[51] A. Meixner and D. J. Sorin. Error detection via online checking of cache coherence with token coherence signatures. In *Proceedings of the IEEE 13th International Symposium on High Performance Computer Architecture*, pages 145–156. IEEE Computer Society, 2007.

[52] G. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8):114–117, 1965.

[53] C. Morin, A.-M. Kermarrec, M. Banatre, and A. Gefflaut. An efficient and scalable approach for implementing fault-tolerant dsm architectures. *IEEE Transactions on Computers*, 49(5):414 –430, 2000.

[54] S. Mukherjee. *Architecture design for soft errors*. Morgan Kauffman, 2008.

[55] S. Mukherjee, M. Kontz, and S. K. Reinhardt. Detailed design and evaluation of redundant multithreading alternatives. In *Proceedings of the 29th International Symposium on Computer Architecture*, pages 99–110. ACM, 2002.

[56] R. Naseer and J. Draper. Dec ecc design to improve memory reliability in sub-100nm technologies. In *15th IEEE International Conference on Electronics, Circuits and Systems*, pages 586 –589. IEEE Computer Society, 2008.

[57] S. R. Nassif, N. Mehta, and Y. Cao. A resilience roadmap. In *Design, Automation, and Test in Europe*, pages 1011–1016, 2010.

[58] M. Nicolaidis and R. Duarte. Fault-secure parity prediction booth multipliers. *IEEE Design Test of Computers*, 16(3):90 –101, 1999.

[59] M. Nicolaidis, R. Duarte, S. Manich, and J. Figueras. Fault-secure parity prediction arithmetic operators. *IEEE Design Test of Computers*, 14(2):60 –71, 1997.

[60] N. Oh, P. Shirvani, and E. McCluskey. Control-flow checking by software signatures. *IEEE Transactions on Reliability*, 51(1):111 –122, 2002.

[61] N. Oh, P. Shirvani, and E. McCluskey. Error detection by duplicated instructions in super-scalar processors. *IEEE Transactions on Reliability*, 51(1):63 –75, mar 2002.

[62] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang. The case for a single-chip multiprocessor. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 2–11. ACM, 1996.

[63] A. Parashar, A. Sivasubramaniam, and S. Gurumurthi. Slick: slice-based locality exploitation for efficient redundant multithreading. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, volume 34-5, pages 95–105. ACM, 2006.

[64] D. A. Patterson, P. Garrison, M. Hill, D. Lioupis, C. Nyberg, T. Sippel, and K. V. Dyke. Architecture of a vlsi instruction cache for a risc. In *Proceedings of the 10th Annual International Symposium on Computer Architecture*, pages 108–116. ACM, 1983.

[65] A. Pour and M. Hill. Performance implications of tolerating cache faults. *IEEE Transactions on Computers*, 42(3):257 –267, 1993.

[66] M. Prvulovic, Z. Zhang, and J. Torrellas. Revive: cost-effective architectural support for rollback recovery in shared-memory multiprocessors. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 111–122. IEEE Computer Society, 2002.

[67] P. Racunas, K. Constantinides, S. Manne, and S. Mukherjee. Perturbation-based fault screening. In *IEEE 13th International Symposium on High Performance Computer Architecture*, pages 169 –180, 2007.

[68] B. M. Rajesh Venkatasubramanian, J.P. Hayes. Low-cost on-line fault detection using control flow assertions. In *Proceedings of the 9th IEEE On-Line Testing Symposium*, pages 137–143. IEEE Computer Society, 2003.

[69] M. Rashid and M. Huang. Supporting highly-decoupled thread-level redundancy for parallel programs. In *Proceedings of the 14th International Symposium on High Performance Computer Architecture*, pages 393–404. IEEE Computer Society, 2008.

[70] S. K. Reinhardt and S. S. Mukherjee. Transient fault detection via simultaneous multithreading. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 25–36. ACM, 2000.

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

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

[73] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, D. I. August, and S. S. Mukherjee. Design and evaluation of hybrid fault-detection systems. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 148–159. IEEE Computer Society, 2005.

[74] S. . F. Remarks. Selse 2 reverie. 2006.

[75] D. Roberts, N. S. Kim, and T. Mudge. On-chip cache device scaling limits and effective fault repair techniques in future nanoscale technology. In *10th Euromicro Conference on Digital System Design Architectures, Methods and Tools*, pages 570 –578, 2007.

[76] G. R. Roelke. *Fault and Defect Tolerant Computer Architectures: Reliable Computing With Unreliable Devices*. PhD. Thesis, 2006.

[77] A. Ros, M. E. Acacio, and J. M. García. A scalable organization for distributed directories. *Journal of Systems Architecture*, 56(2-3):77–87, 2010.

[78] E. Rotenberg. Ar-smt: A microarchitectural approach to fault tolerance in microprocessors. In *Proceedings of the 29th Annual International Symposium on Fault-Tolerant Computing*, pages 84–. IEEE Computer Society, 1999.

[79] S. Rusu, H. Muljono, and B. Cherkauer. Itanium 2 processor 6m: higher frequency and larger l3 cache. *IEEE Micro*, 24(2):10 – 18, 2004.

[80] R. Sasanka, S. V. Adve, Y.-K. Chen, and E. Debes. The energy efficiency of cmp vs. smt for multimedia workloads. In *Proceedings of the 18th annual international conference on Supercomputing*, pages 196–206. ACM, 2004.

[81] P. P. Shirvani and E. J. McCluskey. Padded cache: A new fault-tolerance technique for cache memories. In *Proceedings of the 17TH IEEE VLSI Test Symposium*, pages 440–. IEEE Computer Society, 1999.

[82] P. Shivakumar, M. Kistler, S. W. Keckler, D. Burger, and L. Alvisi. Modeling the effect of technology trends on the soft error rate of combinational logic. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, pages 389–398. IEEE Computer Society, 2002.

[83] J. C. Smolens, B. T. Gold, B. Falsafi, and J. C. Hoe. Reunion: Complexity-effective multicore redundancy. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 223–234. IEEE Computer Society, 2006.

[84] J. C. Smolens, B. T. Gold, J. Kim, B. Falsafi, J. C. Hoe, and A. G. Nowatzyk. Fingerprinting: Bounding soft-error-detection latency and bandwidth. *IEEE Micro*, 24(6), 2004.

[85] A. Sodani and G. S. Sohi. Dynamic instruction reuse. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 194–205. ACM, 1997.

[86] G. S. Sohi. Cache memory organization to enhance the yield of high performance vlsi processors. *IEEE Transactions on Computers*, 38:484–492, April 1989.

[87] D. Sorin, M. Hill, and D. Wood. Dynamic verification of end-to-end multiprocessor invariants. In *Proceedings of International Conference on Dependable Systems and Networks*, pages 281 – 290, 2003.

[88] D. J. Sorin. *Fault Tolerant Computer Architecture*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2009.

[89] D. J. Sorin, M. M. K. Martin, M. D. Hill, and D. A. Wood. Safetynet: improving the availability of shared memory multiprocessors with global checkpoint/recovery. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 123–134. IEEE Computer Society, 2002.

[90] L. Spainhower and T. A. Gregg. Ibm s/390 parallel enterprise server g5 fault tolerance: a historical perspective. *IBM Journal of Research and Development*, 43:863–873, September 1999.

[91] J. Stathis. Physical and predictive models of ultrathin oxide reliability in cmos devices and circuits. *IEEE Transactions on Device and Materials Reliability*, 1(1):43 –59, Mar. 2001.

[92] P. Subramanyan. *Efficient Fault Tolerance in Chip Multiprocessors Using Critical Value Forwarding*. PhD. Thesis, 2010.

[93] F. Sultan, L. Iftode, and T. Nguyen. Scalable fault-tolerant distributed shared memory. In *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*. IEEE Computer Society, 2000.

[94] D. Sunada, M. Flynn, and D. Glasco. Multiprocessor architecture using an audit trail for fault tolerance. In *Proceedings of the 29th International Symposium on Fault-Tolerant Computing*, pages 40–. IEEE Computer Society, 1999.

[95] K. Sundaramoorthy, Z. Purser, and E. Rotenberg. Slipstream processors: improving both performance and fault tolerance. In *Proceedings of the 9th Conference on Architectural Support for Programming Languages and Operating Systems*, pages 257–268. ACM, 2000.

[96] D. Sánchez, J. L. Aragón, and J. M. García. Evaluating dynamic core coupling in a scalable tiled-cmp architecture. In *Proceedings of the 7th International Workshop on Duplicating, Deconstructing, and Debunking. In conjunction with ISCA'08*, 2008.

[97] D. Sánchez, J. L. Aragón, and J. M. García. Extending srt for parallel applications in tiled-cmp architectures. In *14th IEEE Worshop on Dependable Parallel, Distributed and Network-Centric Systems, in conjuction with IEEE International Symposium on Parallel Distributed Processing*, pages 1 –8, 2009.

[98] D. Sánchez, J. L. Aragón, and J. M. García. A log-based redundant architecture for reliable parallel computation. In *17th International Conference on High Performance Computing*, 2010.

[99] D. Sánchez, J. L. Aragón, and J. M. García. Repas: Reliable execution for parallel applications in tiled-cmps. In *Proceedings of the 15th International European Conference on Parallel and Distributed Computing*, pages 321–333, 2009.

[100] D. Sánchez, Y. Sazeides, J. L. Aragón, and J. M. García. An analytical model for the calculation of the expected miss ratio in faulty caches. In *17th International On-Line Testing Symposium*, 2011.

[101] Y. Taur. CMOS design near to the Limit of Scaling. *IBM Journal of Research and Development*, 46(2/3):213–222, 2002.

[102] M. B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, J.-W. Lee, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal. The raw microprocessor: A computational fabric for software circuits and general-purpose programs. *IEEE Micro*, 22(2):25–35, 2002.

[103] J. M. Tendler, J. S. Dodson, J. S. Fields, H. Le, and B. Sinharoy. Power4 system microarchitecture. *IBM Journal of Research and Development*, 46:5–25, 2002.

[104] S. Thoziyoor, J. H. Ahn, M. Monchiero, J. B. Brockman, and N. P. Jouppi. A comprehensive memory modeling tool and its application to the design and analysis of future memory hierarchies. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, pages 51–62. IEEE Computer Society, 2008.

[105] W. J. Townsend, J. A. Abraham, and E. E. Swartzlander, Jr. Quadruple time redundancy adders. In *Proceedings of the 18th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, pages 250–. IEEE Computer Society, 2003.

[106] N. Verma and A. Chandrakasan. A 256 kb 65 nm 8t subthreshold sram employing sense-amplifier redundancy. *IEEE Journal of Solid-State Circuits*, 43(1):141 –149, 2008.

[107] T. N. Vijaykumar, I. Pomeranz, and K. Cheng. Transient-fault recovery using simultaneous multithreading. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 87–98, 2002.

[108] N. J. Wang and S. J. Patel. Restore: Symptom based soft error detection in microprocessors. In *Proceedings of the 2005 International Conference on*

Dependable Systems and Networks, pages 30–39. IEEE Computer Society, 2005.

[109] D. L. Weaver and T. Germond. *The SPARC Architecture Manual*. SPARC International, Inc., 1992.

[110] T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos. Mechanisms for store-wait-free multiprocessors. In *Proceedings of the 36th Annual International Symposium on Computer architecture*, pages 266–277. ACM, 2007.

[111] C. Wilkerson, H. Gao, A. R. Alameldeen, Z. Chishti, M. Khellah, and S.-L. Lu. Trading off cache capacity for reliability to enable low voltage operation. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, pages 203–214, 2008.

[112] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The splash-2 programs: characterization and methodological considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–36. ACM, 1995.

[113] M. Yamaoka, K. Osada, R. Tsuchiya, M. Horiuchi, S. Kimura, and T. Kawahara. Low power sram menu for soc application using yin-yang-feedback memory cell technology. In *Symposium on VLSI Circuits*, pages 288 – 291, 2004.

[114] S. B. Yao. Approximating block accesses in database organizations. *ACM Communications*, 20:260–261, April 1977.

[115] L. Yen, J. Bobba, M. R. Marty, K. E. Moore, H. Volos, M. D. Hill, M. M. Swift, and D. A. Wood. Logtm-se: Decoupling hardware transactional memory from caches. In *Proceedings of the 19th International Symposium on High Performance Computer Architecture*, pages 261–272, 2007.

[116] S. Zafar, B. Lee, J. Stathis, A. Callegari, and T. Ning. A model for negative bias temperature instability (nbti) in oxide and high kappa. In *Symposium on VLSI Technology*, pages 208 – 209, 2004.

[117] K. Zhang, U. Bhattacharya, Z. Chen, F. Hamzaoglu, D. Murray, N. Vallepalli, Y. Wang, B. Zheng, and M. Bohr. Sram design on 65nm cmos technology with integrated leakage reduction scheme. In *Symposium on VLSI Circuits*, pages 294 – 295, 2004.

[118] J. Ziegler and W. A. Lanford. The effect of sea level cosmic rays on electronic devices. *Journal of Applied Physics*, 52:4305–4312, 1981.

[119] J. F. Zielger and H. Puchner. *SER-History, Trends and Challenges*. Cypress Semiconductor Corporation, 2004.