UNIVERSIDADE DE SANTIAGO DE COMPOSTELA

Departamento de Electrónica e Computación

Centro de Investigación en Tecnoloxías da información (CITIUS)

PhD Dissertation

NEW HARDWARE SUPPORT FOR TRANSACTIONAL MEMORY AND PARALLEL DEBUGGING IN MULTICORE PROCESSORS

Author:

Lois Orosa Nogueira

Phd Advisors:

Javier Díaz Bruguera

Elisardo Antelo Suárez

Santiago de Compostela, June 2013

Javier Díaz Bruguera, Profesor Catedrático de Universidade da Área de Arquitectura e Tecnoloxía de Computadores da Universidade de Santiago de Compostela

Elisardo Antelo Suárez, Profesor Titular de Universidade da Área de Arquitectura e Tecnoloxía de Computadores da Universidade de Santiago de Compostela

FAN CONSTAR:

Que a memoria titulada **NEW HARDWARE SUPPORT FOR TRANSACTIONAL MEMORY AND PARALLEL DEBUGGING IN MULTICORE PROCESSORS** foi realizada por **D. Lois Orosa Nogueira** baixo a nosa dirección no Departamento de Electrónica e Computación e no Centro Singular de Investigación en Tecnoloxías da Información (CITIUS) da Universidade de Santiago de Compostela, e constitue a Tese que presenta para optar ao grado de Doutor pola Universidade de Santiago de Compostela.

Santiago de Compostela, Xuño 2013

Javier Díaz Bruguera
Codirector da tese

Elisardo Antelo Suárez
Codirector da tese

Lois Orosa Nogueira
Autor da tese

Aos meus pais

Everything that can be invented has been invented.
Charles H. Duell, U.S. patent office, 1899

It would appear that we have reached the limits of what it is possible to achieve with computer technology, although one should be careful with such statements, as they tend to sound pretty silly in 5 years.
John Von Neumann, 1949

O verdadeiro heroísmo está en transformar os desexos en realidades e as ideas en feitos.
Castelao

Acknowledgements

Five years ago I gave a radical turn to my life: I quit a stable job in a private company to start this research adventure. The way was long, it had ups and downs, but at the end, it was worth it. It was an invaluable experience that changed and marked me forever, personally and professionally, and I would not have been able to do it alone. It is at this point that I want to thank all the people that made this thesis possible.

First of all, and the most important, I want to thank my advisors Elisardo Antelo and Javier Bruguera. They trusted in me from the beginning of this adventure, and we start and finish this trip together. Without them, this thesis would simply not be possible. Their constant support was essential to finish it, as also the advices and motivating talks of Elisardo. Thank you very much.

To Professor Josep Torrellas, who supervised my work in my stay in the University of Illinois at Urbana-Champaign, and to all the members of his group for their kind welcome. Special thanks to Shanxiang Qi and Norimasa Otsuki for the good work environment and the intriguing discussions.

To the people of IBM R&D research Lab in Haifa for their friendly welcome during my HiPEAC internship there. My acknowledgements goes specially to Olga Golovanevsky, Marina Biberstein and Bilha Mendelson for their daily support and the motivating work made there.

To Recore Systems, specially to Gerard Rauwerda, for trusting me to perform a very engaging project during my HiPEAC internship, and to John Donker, Jordy Potman and Eduard Fernández for their every day support and knowledge, which enriched my stay.

I also want to express my gratitude with the funding institutions. The work related to this PhD thesis was partially supported by the Spanish Ministry of Science and Education under Project TIN2007-67537-C03-01. I also wish to thank the European Network of Excellence on

High Performance and Embedded Architecture and Compilation (HiPEAC) for the funding that allowed me to perform my internships in Haifa and Enschede during my thesis period.

Fora do ámbito profesional, tamén quero agradecer á xente que estivo ao meu arredor todos estes anos deixando un recordo indeleble da miña etapa de tese e servindo como válvula de escape cando mais o necesitaba. Grazas aos meus amigos, compañeiros de piso, compañeiros de departamento, á xente que fixo mais levadeiras as miñas estancias no estranxeiro e a esas persoas especiais que pasaron pola miña vida. Grazas a Xacobo&Marta, Marcos, Jose, Rodrigo, Rafa, Antón, Manolo, Nerea&Manolo, Ricardo, Fran, Saul e Ángela por tódolos momentos vividos en Compostela; grazas a Antón por todo o tempo e aventuras compartidas; grazas a Carlos, Manolo e Yolanda por ser os compañeiros de piso perfectos; grazas a Ale, Juli, Emi, Nino, Aitor, Miguel, Gabi, Víctor, Jose, Manu&Patri, Paula, Tania, Deborah e demais tropa viguesa polos momentos vividos nas nosas xuntanzas; grazas aos compañeiros de ESF por tanta implicación e rebeldía; grazas a Cris, Fer, Enrique(s) e demais xente do departamento por compartir preocupacións, sufrimentos e festas; thanks to Norimasa for making my stay at Urbana-Champaign very special; thanks to Vladimir, Waldemar and Per for making my stay in Haifa an experience that I will never forget; thanks to Jarkko, Eduard, Aria-Lena and Stefania for being my happiness during my stay in Enschede; grazas a Sandra por axudarme a tomar a decisión de comezar esta aventura, e por apoiarme nos meus primeiros anos de tese; grazas a Sol por estar aí nesta última etapa; e grazas a moitos outros que non nomeo pero que igualmente foron parte fundamental para o meu desenvolvemento persoal e profesional.

Quero dar tamén as grazas á miña familia, sobre todo aos meus primos Marcos, Telmo, Paula&Alberto , Marta, Lucía, Laura, Alex, e Noa, e moi especialmente aos meus pais Suso e Carmiña, por apoiarme sempre. Ademais quero ter un recordo moi especial coa miña avoa Dolores. Por último, quero dar tamén as gracias ao meu irmán Adrián, a Maca, e ao novo recén chegado, o meu sobriño Ían.

Moitas grazas a todos! Thank you very much!

Santiago de Compostela, June 2013

Contents

List of Figures

List of Tables

Preface

In the multicore era, parallel programming is becoming a must for general purpose programmers. However, the parallelization of programs is not intrinsically intuitive and prone to errors. To face these drawbacks, new tools have arisen to make this task easier by providing new programming models and debugging tools. Usually, these new tools add complexity that needs to be addressed with hardware support to achieve a good performance. Transactional Memory and data race detection are two of the most popular.

Transactional Memory (TM) is a software abstraction to make parallel programming easier, by providing an abstraction less prone to errors than locks. TM provides an speculative synchronization mechanism to the programmer, who has to enclose critical sections in transactions which will be executed atomically and in isolation by the TM system. Furthermore, transactions are executed in parallel and speculatively, so it is pretty common to use hardware support to accelerate the system.

A data race occurs when two or more threads access the same shared variable without the proper synchronization. Data races can produce errors which are very difficult to debug, because it is usual that the error manifests itself much later than the actual race is produced. Therefore, to efficiently detecting these bugs has become very important, and also for these tools, it is not strange to use hardware support to not degrade the performance of the system.

Among the hardware resources for accelerating this kind of tools, one of the most generally used in research papers, and therefore, with a lot of potential to be included in future general purpose processors, are signatures. Signatures are a fixed piece of hardware that can host an unbounded number of addresses in a bounded space. To do that, each address is hash encoded and inserted in the signature, which may produce aliasing among address. This leads to the possibility of reporting false positives when the signature checks for the ownership of an address. However, it never reports false negatives.

This thesis contributes with new hardware support for TM and data race detection in multicore processors. The hypothesis to build this thesis are:

– Hardware signatures are not optimized for the many applications and tools that use them (very different ones in the same system).

– Signatures can be used in a large number of applications related with parallel programming of multicore processors, and some of them are unexplored.

– Signatures are a promising hardware resource because of their efficiency, but they have drawbacks that have not been explored: they are not flexible to adapt to the different requirements of the applications and tools that may require them in a multicore processor.

Under this hypothesis, we set up and perform experiments to address these observations and problems. In Chapter 3 we configure a Hardware Transactional Memory (HTM) system where signatures are part of the hardware support and we propose a new hardware filter based on minor modifications of the hardware, which allows a considerable reduction of the signature size either their false positive rate (we call this filter CFM-TM). Under certain circumstances, the performance of the system is also significantly improved. We observe from this work that to optimize the resources and the false positive rate, we require signatures with different sizes.

In Chapter 4 we build the first hardware asymmetric data race detector (which also tolerates these races), called Pacman [1]. Asymmetric data races are a very common type of data race that may cause dangerous concurrent bugs, and that until this work, have only been explored as a software approach. The hardware support of our detector is essentially based on a centralized module of hardware signatures. We demonstrate that Pacman introduces negligible slowdowns in the system, and that it is able to efficiently detect and tolerate asymmetric data races.

Chapter 5 and Chapter 6 propose a novel hardware signature module (called *FlexSig*) that solves some of the problems that we found when building the previous tools for multicore architectures based on signatures. Specifically, we design a signature module that can host a large number of signatures when there is a high demand of signatures, and it can also achieve

[1]This work was developed at the University of Illinois at Urbana-Champaign in collaboration with the members of the I-ACOMA group.

a very low false positive rate when the demand of signatures is modest. We explore several strategies to allocate signatures in *FlexSig* to adapt to the different characteristics of the tools and applications that use them.

Summarizing, we optimized the use of signatures in a HTM system by introducing our CFM-TM filter, we developed a new debugging tool with signatures as main hardware support, and we built a new hardware signature module that allows a great flexibility in the size and number of signatures allocated. This fits in a real scenario represented by a general purpose multicore processor executing a wide range of signature-demanding applications and tools. We have gathered the contributions in the next publications:

Lois Orosa, Javier D. Bruguera, and Elisardo Antelo. **A Cache Filtering Mechanism for Hardware Transactional Memory Systems Decoupled from Caches**. In *XX Jornadas de Paralelismo*, A Coruña (Spain), September 2009.

Shanxiang Qi, Norimasa Otsuki, Lois Orosa, Abdullah Muzahid, and Josep Torrellas. **Pacman: Tolerating Asymmetric Data Races with Unintrusive Hardware**. In *High Performance Computer Architecture (HPCA)*, 2012 IEEE 18th International Symposium on, pages 1 –12, feb. 2012.

Lois Orosa, Elisardo Antelo, and Javier D. Bruguera. **Flexsig: Implementing Flexible Hardware Signatures**. *ACM Trans. Archit. Code Optim.*, 8(4):30:1–30:20, January 2012.

Lois Orosa, Javier D. Bruguera and Elisardo Antelo. **Asymmetric Allocation in a Flexible Signature Module for Multicore Processors**. *Submitted.*

Chapter 1

Introduction

Sequential single-core wide superscalar processors were the dominant architecture in commodity products during many years for their capacity of improving performance by just scaling the clock frequency (with contributions from both technological and pipeline depth scaling), and keeping the cycles per instruction. The software for these processors is an ordered sequence of instructions, being easy to write and reason about it, and therefore, the performance scales naturally with clock frequency.

Parallel computers were already used many years ago, but mainly for scientific applications. The programming skills required were very high because of the difficulty of debugging and reasoning about parallel programs.

Nowadays, parallel architectures have become a mainstream research topic among industry and academia for general purpose computing, specially focusing on integrating many processor units in the same chip (multicore processor). The interest in multicore architectures arose mainly for two reasons: the Instruction Level Parallelism (ILP) was no longer cost effective (ILP wall), and the power consumption resulting from increasing the clock frequency started to become a limiting effect (Power Wall). Multicore processors encourage thread parallelism, that scales better than ILP (if the applications are efficiently parallelized), and also allows us to maintain the clock frequency in moderate rates and improve performance by augmenting the number of cores in the chip. However, there is another problem that could limit the multicore scaling: not all the transistors of the chip can be powered at the same time because of power limitations; the area of the chip that can not be powered is known as dark silicon, and with each new generation of processors (which imply more transistors in the same area), the percentage of the chip that can actively be used is dropping exponentially.

The non deterministic execution of parallel programs in most of the parallel architectures, as well as the previous experience programming the old parallel machines, teach us that parallel programming is not an easy task. One of the main challenges in this field is to make parallel programming affordable by most of the general purpose programmers, which requires new tools, programming models, debugging tools, etc, and in most of the cases, hardware to accelerate and support them.

This introductory chapter is organized as follows: we discuss issues related to parallel architectures in Section 1.1, Section 1.2 exposes some aspects about parallel programming, Section 1.3 explains the most common issues in parallel programming, Section 1.4 introduces Transactional Memory and finally, Section 1.5 is devoted to the discussion of signatures, an essential building block for the schemes proposed in this thesis.

1.1 Parallel Architectures

1.1.1 Why Parallel Architectures?

Moore's Law [108] predicts that the number of transistors doubles approximately every two years, and along the time, this law proved to be pretty accurate. The design goal was to drive the frequency up and add more transistors to the chip. However, in the last years the increased frequency also augmented the power dissipation beyond the capacity of the cooling techniques (the power wall). In this context, to have several cores in the same chip without scaling clock frequency became the best option to continue increasing the performance with a reasonable power consumption.

Multicore processors are composed by several cores in the same chip. In this thesis, we will focus on shared memory multicore processors, which are composed by several processing cores (with private cache), a last level shared cache memory and an on chip interconnection network. Usually, the core uses its local private memory as caches to keep private and shared data locally (for performance reasons), and the system maintains the data coherent with a cache coherent protocol. This shared memory model is widely used in parallel programming because it is in principle easy and intuitive.

The inspiration of the multicore processors comes from the Symmetric Multi-Processors (SMPs) used in High Performance Computing (HPC). In 1962 it was implemented the first

SMP (BURROUGHS D825 [9]), and since then, these machines have been used to solve very complex simulations and problems, such as scientific applications in many different domains (bio-sciences, computer engineering, physics, chemical engineering, etc). They are composed of many processors (hundreds or thousands) connected with high bandwidth networks. The idea of multicore processors is to build a SMPs in a single chip, which allows a very low power consumption and very fast inner-chip communication. The biggest microprocessor manufacturers already have multicore processors with a few cores in the market, and prototypes and future commercial chips with tens of cores [59]. Notice that the SMPs used nowadays in HPC usually are composed of many multicore processors connected among them.

There are two trends in parallel architectures: homogeneous architectures and heterogeneous architectures. Homogeneous processors are composed of many replicas of the same core, like many of the multicore processors in the market. However, heterogeneous architectures also have a big importance, as many chips are composed of cores of different natures (it is very common to integrate Central Processing Units (CPUs) and Graphics Processing Units [23] (GPUs) in the same chip). Examples of heterogeneous architectures are the IBM Cell [79] (composed of one Power architecture core and eight specialized co-processors), Nvidia's Tegra [117] (that consists of eight independent processors for graphics, video encode and decode, image processing, audio processing, power management, and general-purpose functions) or AMD's Fusion [5] (CPU and GPU).

1.1.2 Levels of Parallelism

The most well known kinds of parallelism are Instruction Level Parallelism (ILP), Thread Level Parallelism (TLP) and Data Level Parallelism (DLP), which we describe below.

Instruction Level Parallelism (ILP) exploits parallelism at instruction level, by using pipelining to overlap the execution of instructions, and superscalar out of order execution. Different instructions can be executed simultaneously if they are not using the same functional units (which is the reason to replicate some of them), if there are no data dependencies and provided that branches are predicted in a correct way most of the time. One of the main advantages of the ILP is that it is transparent to the programmer, because the parallelism is obtained automatically with processor and compiler techniques. However, the limit on how much ILP can be achieved seems to already have reached a saturation point [174] and is one of the reasons of the architecture shift from single core architectures to multicore. Almost all current general purpose processors have implemented some kind of ILP.

Thread Level Parallelism (TLP) exploits parallelism at thread level. A program is structured in threads by the programmer (unlike ILP) that executes different tasks composed of many instructions. With the popularity of the multicore processors, these type of parallelism began to have more relevance. TLP is orthogonal and compatible with ILP.

Data Level Parallelism (DLP) consists of performing the same operation over multiple data simultaneously (vector machines, array computers, vector extensions in modern microprocessors, etc.). One particular architecture to exploit DLP are GPUs. GPUs are composed of many lightweight cores (100s), and unlike multicore processors, GPUs use lightweight threads, the single thread performance is very poor, and are good at parallel, arithmetically intense, streaming memory problems. Originally they were designed for graphics processing, but nowadays they are also used to solve complex computing parallel problems, which are know as GPGPU (General Purpose on Graphics Processing Units).

1.1.3 Shared Memory Multicore Processors

The base architecture over which this thesis was developed is based on Shared Memory Multicore processors [53] [18]. They are homogeneous architectures designed to exploit thread level parallelism and run multiple threads simultaneously in the different processing cores.

Usually, each core in the chip has one or two levels of private cache to improve the performance of data access. Furthermore, they also have a last level cache shared by all the cores which is distributed along all the chip. To access the shared resources efficiently, the cores are connected with a high bandwidth on-chip network. Shared data is kept coherent among cores through the cache coherence protocol, which can be implemented very effectively by the low latency interconnection network. Figure 1.1 depicts a basic instance of a simple shared multicore architecture.

Furthermore, the memory consistency model is also very important. This is a conceptual model for the semantics of memory operations to allow them to correctly use the shared memory (more details in Section 1.1.3). This model should be simple from the point of view of the programmer, and at the same time should have a good performance.

Cache Coherence

Despite being easy to program, shared memory architectures also have some drawbacks, resulting in more hardware complexity. These architectures usually support the caching of private and shared data. Private data are used by a single core, while shared data can be used

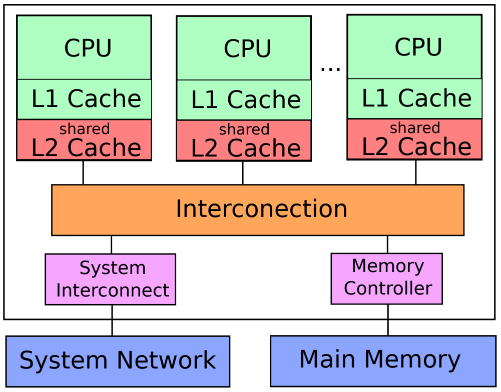


Figure 1.1: Basic strucure of a shared memory multicore processor.

by all cores. When private data is cached, a copy of the data is migrated to cache, reducing the access time in future accesses. Shared data can also be in multiple private caches for reducing the average access time, but at the same time two different cores could end up using two different values for the same data address. This situation introduces the problem of the cache coherence [161].

In modern multiprocessors, the cache coherence protocols are implemented in hardware, and there are two basic classes:

– Snoopy protocols: the caches are accessible by some broadcast medium (a bus, for instance), and all cache controllers monitor (or snoop) the medium to determine the appropriate actions to take in response to memory operations issued by other cores.

– Directory-based protocols: The status of the cache lines is maintained in a directory, which can be hosted in the main memory or distributed among the caches with a linked list. It has slightly more overhead than snoopy protocols but it can scale to larger core counts.

A simple and very common instance of a cache coherence protocol is **MESI** [1]. In this protocol each cache line can be in four different states:

[1]Industrial implementations add some additional states to adapt for specific situations. For instance, the MESIF and MOESI protocols are used by Intel and AMD respectively, for optimizing NUMA implementations.

- **M**odified: the line has been modified in the cache, and it is the only copy in local caches. This data is inconsistent with the copy in main memory, so the protocol should update it when the cache line is evicted.
- **E**xclusive: the line is present only in that cache and it is consistent with main memory.
- **S**hared: the line is unmodified (with respect to the copy in main memory), and may have other unmodified copies in other local caches. All copies of the cache line are consistent with main memory.
- **I**nvalid: the copy of the line is invalid.

The coherency states are maintained through communication among caches. Figure 1.2 shows an instance of the transition states for a MESI snoopy protocol. The local processor is capable of performing the following actions: to load data in the local cache (Load Rd), to store data into a local cache line (Local Wr) and to replace one line for another one (Flush). The local actions can result in the bus activity seen by the other cache controllers (Remote Rd and Remote Wr).

The basic transitions among the MESI states are shown in Figure 1.2. When a local read request arrives at a cache for a line in M, E or S states, the controller provides the line, but if it is in I state, it has to verify that it is not in M in a remote cache, in which case this cache with the data in M state has to write back the data in memory, and change the state to S. Finally, the cache that originally requested the data gets a copy from memory or from other caches with the data in S or E state.

When a local write request arrives in a cache for a line in M or E state, the cache modifies the data locally. If the line is in S state, the cache must notify to other caches that might contain the line in S, E or M states that they must invalidate the line. Then the data may be locally modified. If the line is in the I state, the cache must notify any other caches that might contain the line in the S, E or M states that they must invalidate the line. If the line is in another cache in the M state, that cache must either write the data to main memory or supply it to the requesting cache. If at this point the cache does not yet have the line locally, the line is read from the main memory before being modified in the cache. After the data is modified, the cache line is in state M.

The MESI protocol can be implemented with a snoopy protocol, as in the previous example, or with a directory. The implementation of MESI with a directory introduces new

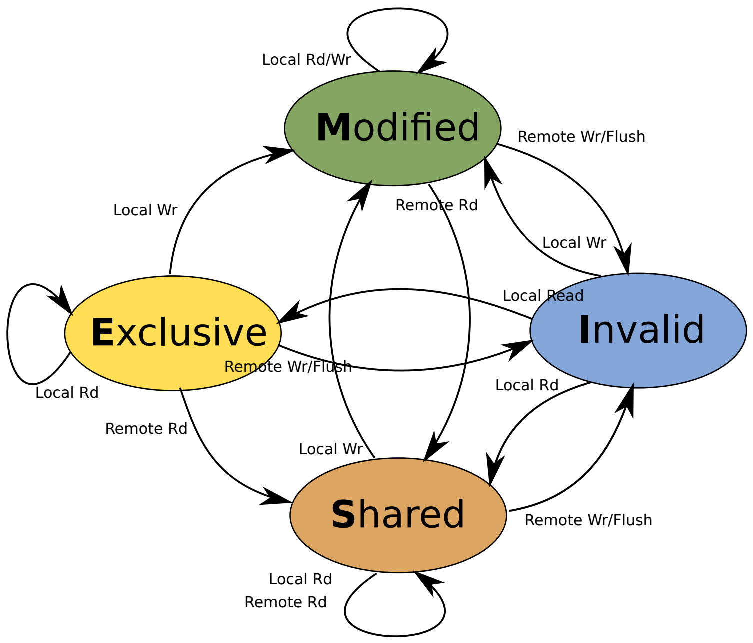


Figure 1.2: Transition states in a MESI cache coherence protocol.

intermediate states in the protocol because there is not an atomic bus over which to make atomic operations. Also, a directory consumes less bandwidth than a broadcast protocol.

Consistency Models

One of reasons why parallel systems are not intuitive is the memory consistency model [54]. In a sequential processor, it is easy to reason that a load operation reads the most recent written value. However, a parallel computer with shared memory has several threads accessing to data (issuing reads and writes) independently and concurrently. In this scenario it might be very complex to reason about the value that should be retrieved from a read. Memory consistency models have been developed to specify what values may be returned by a read, because the operations are not fully ordered (unlike in sequential systems).

The **sequential consistency model** [86] is the most intuitive, and straightforward. It requires that the result of any execution be the same as if the memory accesses executed by each processor were kept in order and the accesses among different processors were arbitrarily interleaved. Although sequential consistency presents a simple programming paradigm, its implementation reduces the performance, specially when the number of cores in the processor is high. There are some examples of parallel architectures with a sequential consistency model [29] [90].

Relaxing the consistence memory model is the way to improve performance and to reduce hardware overhead. The idea is to allow reads and writes to update main memory or a shared cache out of program order, and to use synchronization operations (introduced by the programmer) to enforce ordering, so that a synchronized program behaves as a sequentially consistent processor. There are a variety of relaxed models that are classified by the relaxed read and write orderings. Sequential consistency requires maintaining all possible orderings: R->W (a write can not be executed until preceding reads have been completed), R->R (a read can not be executed until preceding reads have been completed), W->R (a read can not be executed until preceding writes have been completed) and W->W (a write can not be executed until preceding writes have been completed). The **total store ordering (TSO)** [74] is characterized for relaxing the W->R; because this ordering retains ordering among writes, many programs that operate under sequential consistency operate under this model without extra synchronization. The **partial store order (PSO)** [74] relaxes the W->W ordering. The **release consistency (RC)** [55] relaxes R->W and R->R.

By relaxing these orderings, the processor can achieve performance advantages over sequential consistency at the cost of more programming and debugging effort.

1.2 Parallel Programming

Architectures with many cores in the same chip scale their performance with the number of cores only if the software executed is parallelized efficiently. For parallel programming success among general purpose programmers, it has to be easy to code, debug, and easily understood by an average programmer (not an expert in parallel programming).

The natural way of thinking about algorithms to solve problems is sequentially. The solution of a problem is always envisioned as a series of steps that are performed one after the other, and many times parallelism is not an option for many programmers. Changing the mentality into parallel reasoning is a challenge that should be driven through new software abstractions and tools (easy to understand), and efficient parallel debugging tools.

However, programs usually have sections of code that can not be parallelized. The Amdahl law [7] states that the theoretical maximum speedup of a program using multiple cores is limited by the time needed for the sequential fraction of the program (and the overhead introduced by parallelization, such as synchronization or message latencies). Therefore, the performance of a parallel program does not depend only on the hardware architecture, but also on the amount of parallelizable sections of code, and in the ability of the programmer to identify and efficiently parallelize these sections.

To get an idea of why parallel programming is difficult, we will expose some ideas that are advocated by several authors [166][56]. Below, we depict some historical facts which impacted parallel programing.

- The community concentrated on improving performance instead of reducing software development cost.
- Early parallel machines had poor support for communication among processors, and much of the compiler and programming effort was to reduce communications cost, rather than high level ideas about how to simplify applications and algorithms.
- The emphasis was to produce simple modifications to sequential languages, such as adding libraries, rather than thinking in parallelism from scratch.

These could be some reasons that explain the current state of the art in parallel programming. Furthermore, parallel programming is also intrinsically difficult because:

- **Finding parallelism might be difficult and add programing complexity**. There are some steps needed to create a parallel program not needed in sequential codes, such as synchronization issues, concurrency detection, task decomposition, load balancing, etc.
- **It is error-prone**. To decompose the problem or algorithm into tasks may lead to errors (main problems are synchronization and non deterministic data races). With several threads running concurrently, and accessing the same data, a new type of bug is introduced: the data races. The threads have to be perfectly coordinated and synchronized to avoid them.
- **Tuning performance**. In the world of multicore processors, things like cache behavior, how the cores are connected, etc. makes a much bigger difference for performance. Due to these issues, programmers may spend a significant time tuning for performance.

- **Future proofing**. In sequential applications the performance increases with the clock frequency, but in parallel applications it is not so easy, because the scaling in the number of cores and the changes of the cache system may affect the previous tuning.

- **Too little knowledge of parallel programming systems**. OpenMP, Erlang, Haskell, X10, Thread Building Blocks (TBB) or Cilk are not widely know by general purpose programmers.

Nowadays, the best known parallel languages/APIs are PThreads [131], MPI [110], OpenMP [120], CUDA [40], OpenCL [119], TBB [141], X10 [149] or Cilk [88]. Despite the fact that concurrent programming may look like a very new paradigm, parallel programming languages has existed from the seventies [65]. The architecture of parallel machines have been changing along the years, and with them also the parallel languages. In the era of vector machines, the parallel languages were only loop annotations; when SIMD processors were the mainstream, data parallel languages becomes popular (CMF, *Lisp, C*, Global Arrays, High Performance Fortran, ...); when shared memory multiprocessors appeared, shared memory models also appeared, such as PosixThreads, OpenMP, etc; and with clusters and Massive Parallel Processors (MPP), message passing become dominant (MPI for instance).

However, there is an alternative to release the programmer from explicit parallelization: automatic parallelization of sequential programs by compilers [21] [162]. Automatic parallelization converts a serial code into parallel code to execute in a shared memory parallel processor. However, fully automatic parallelization of code is still a big challenge because it needs a very complex code analysis, and it has a limited effectiveness without the explicit help of the programmer.

1.2.1 Data Communication

The different threads of an application running in a multicore processor require communication. There are two ways of communication among threads. One is by reading and writing in memory (typically in shared memory multiprocessors), and the other by sending messages (typically in distributed memory multiprocessors) [75]. These two techniques can also be combined and use shared memory communication in a system with distributed memory (Distributed Shared Memory).

Shared Memory Communication

In shared memory parallel programming [57], the communication among threads or processes is done by just shared memory values, since all of them have a common address space. However, to ensure correctness, it is necessary to add mechanisms for correct synchronization. Without proper synchronization among threads, the integrity of data may be destroyed.

The problem of synchronization will be discussed in Section 1.3.1. This model is the most popular in mainstream processors because it is easier to program, and the one that better fits for shared memory multicore processors.

Message Passing

In distributed memory programming [24], the synchronization among processors is done by explicit message passing. Message passing libraries allow the writing of parallel programs for distributed memory systems efficiently. These libraries provide routines to configure the messaging environment and to send and receive packets of data (point to point or collective). The most popular high-level message passing library is MPI (Message Passing Interface). MPI has become the facto standard for message passing parallel programming. This kind of communication has the drawback of being difficult to program because the programmer has to include explicit messages for communication in the code.

1.2.2 Problem Decomposition

Depending on how a problem is decomposed to parallelize it, we distinguish between data and task decomposition.

Data Decomposition

The data parallel model focuses on distributing the data among different computing nodes. It is achieved when the same task is applied over different data in different cores. Data parallelism emphasizes the parallelized nature of the data.

Task Decomposition

Task decomposition refers to dividing the problem in tasks to execute them in different computing nodes (in threads, processes, etc). In general, these processes or threads commu-

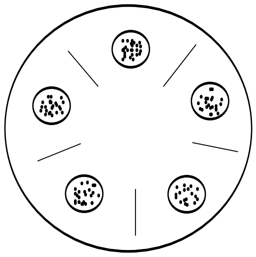


Figure 1.3: Dinning philosophers problem.

nicate with each other (with some form of communication, as showed in Section 1.2.1). Task parallelism emphasizes the parallelized nature of the processing.

1.3 Parallel Programming Issues in Shared Memory

Multicore processors with the shared memory communication model (Section 1.2.1) have some programming issues that have to be considered to program efficiently in parallel. Specifically, synchronization is one of them: it provides mechanisms to programmers for control access to shared data (where there are several threads running concurrently and accessing the same resources) with the aim of avoiding unexpected behaviors caused by thread interleavings not desired by the programmer. Another issue is debugging, as concurrency bugs are difficult to detect and fix so more sophisticated tools are needed to debug programs efficiently. In both cases, hardware support is usually necessary to achieve good performance while keeping correctness.

This thesis is focused on contributing to reduce the overhead of these issues. CFM-TM (Chapter 3) makes a contribution in a speculative synchronization mechanism and Pacman (Chapter 4) in debugging.

1.3.1 Synchronization

The most common synchronization mechanism are locks, which are used for another user level abstractions to build high level synchronization mechanisms (such as semaphores or monitors). To illustrate the synchronization problem, we will expose a classic example (see Figure 1.3):

EXAMPLE: Dining philosophers problem

- Five silent philosophers sit at a table around a bowl of rice.
- A chopsticks is placed between each pair of adjacent philosophers.
- Each philosopher must alternately think and eat.
- Eating is not limited by the amount of rice left: assume an infinite supply.
- A philosopher can only eat while holding both the chopsticks to the left and the chopsticks to the right.
- Each philosopher can pick up an adjacent chopsticks, when available, and put it down, when holding it. These are separate actions: chopsticks must be picked up and put down one by one.

The problem is how to design a discipline of behavior (a concurrent algorithm) so that each philosopher doesn't starve, and they can forever continue to alternate between eating and thinking without a deadlock situation.

If we translate this problem to a shared memory multicore system, we could identify the philosophers as the cores, and the chopsticks as the shared resources. There are several possible solutions for this problem [30] [47], but all of them require some synchronization mechanism to avoid problems like deadlock, one of the most frequent synchronization bugs. In this example, a deadlock could be produced when all the philosophers are frozen with one chopstick in the right hand, and waiting for another chopstick for the left hand (not eating, not thinking).

Locks

Locks are the most common synchronization mechanism [164], used to restrict the concurrent access to some shared data to only one thread at a time. To synchronize threads with them, the critical sections have to be enclosed with locks. Before a thread enters in the critical section, it has to acquire the lock, and when it finishes, it has to release it to allow other threads executing their critical sections. Only one thread can acquire a lock at a time, and the other threads trying to acquire it have to wait.

As locks serialize operations on shared data, the programmer try to either minimize the use of synchronization, or use fine grain locks (multiple locks protect different shared data). With the use of fine grain synchronization, the performance is optimized, but the code becomes prone to errors and difficult to program. On the other hand, if a single lock is used to protect large regions of code (coarse grain synchronization), programming is simpler, but scalability is drastically reduced. It is not appropriate to use coarse grain synchronization with locks for this reason.

The most common bugs due to the use of locks are summarized as follow:

– Deadlock: occurs when a thread is blocked because a resource requested by it is being held by another waiting thread. Figure 1.4 shows an example of deadlock.

– Priority inversion: a high priority thread can not proceed because it is waiting for a lock which is held by a low priority thread.

– Convoying: when multiple threads of equal priority contend repeatedly for the same lock. The threads in a lock convoy do progress, however, each time a thread tries to acquire a lock and fails, it renounces the remainder of its scheduling quantum and forces a context switch. The overhead of this repeated context switching and the underutilization of the quantums degrades performance.

– Livelock: It is similar to deadlock, but the state of the different threads is changing continuously despite there being no progress.

These bugs are very well known, and result in a serious problem for programming productivity.

Locks are built in software, but for performance reasons, they rely on hardware synchronization instructions. The key hardware capability is an uninterruptible instruction capable of atomically retrieving and changing a value. One of these synchronization instructions is the **atomic exchange**, which interchanges a value in a register with a value in memory.

Another common operation is **test-and-set**, which tests the value, and sets it if the value passes the test. For example, we could define an operation that tested for 0 and set the value to 1 (that can be used in a similar way to atomic exchange).

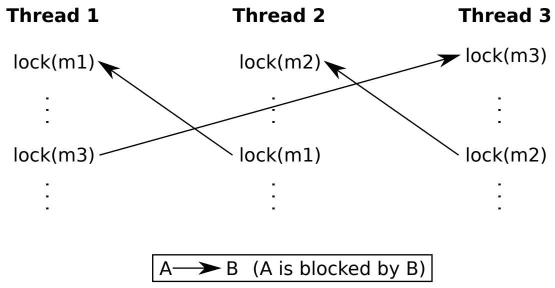


Figure 1.4: Deadlock scenario: the three threads are holding a lock that other thread is trying to get.

Another atomic synchronization primitive is **test-and-increment**: it returns the value of a memory location and atomically increments it (it can be also used in a similar way to atomic exchange).

However, even with these primitives, implementing a single atomic memory operation introduces some challenges, since it requires both a memory read and write in a single, uninterruptible instruction. This requirement complicates the implementation of the coherence, since the hardware can not allow any other operations between the read and the write, and yet it must not deadlock.

An alternative is to have a pair of instructions where the second instruction returns a value from which it can be deduced whether the pair of instructions is effectively atomic if it appears as if all other operations executed by any processor occurred before or after the pair. These pair of instructions includes a special load called **load linked** and a special store called **store conditional**. These instructions are used in sequence: if the contents of the memory location specified by the load linked are changed before the store conditional to the same address occurs, then the store conditional fails. The store conditional is defined to return 1 if it was successful and 0 otherwise.

Semaphores and Monitors

Semaphores and monitors are high-level synchronization mechanisms build on top of locks. Semaphores are pretty close to locks, but add some functionality (for instance, allowing more than one thread access to the critical section). Monitors are a set of multiple routines that are protected by locks and these locks are acquired and released automatically when the routines are used (it is not the responsibility of the programmer). Both are used for the same purposes, the difference is the level of control and abstraction that the programmer has.

Alternatives

The synchronization mechanism based on locks are perfectly valid for modern multicore processors. However, it is difficult to program with them, and they are prone to errors. To improve these approaches, in recent years some proposals like Lock Elision [138] (speculatively removing unnecessary lock-induced serialization in run time) or Transactional Memory [72] have arisen as promising alternatives. We will describe Transactional Memory in detail in Section 1.4.

1.3.2 Debugging Concurrency Bugs in Parallel Programs

Thread interleaving in parallel programs is unpredictable on most of the architectures, which makes it hard to debug because of the difficulty of reproducing a concurrency bug. The production phase of parallel software takes a lot of time, and many errors are not even detected until years of correct execution. In many cases, the time invested in debugging is higher than that invested in coding. Usually these concurrent bugs are showed up only under certain timing conditions, and their effects manifest many instructions after the real bug is produced.

For these reasons, parallel debugging is very important and needs to be supported by appropriate tools to help and accelerate this task. Without them, parallel programming will not become mainstream.

Classical bug techniques (like the "printf" technique) are practical for some easy-to-solve bugs, but with concurrency bugs usually it is very impractical to debug in this way, because of the non-deterministic nature of parallel software.

Most previous concurrency bug detection schemes were focused on detecting data races [33] [48] [151] [130], deadlocks [22] [48] [151] or atomicity violations [92] [94]. There are also other approaches to support and help debugging, such as deterministic replay [107] [10] [114] [115] [181] [128] [34], which can replay a bug deterministically, or parallel architectures that implement more easy to understand memory models, such as BulkSC [29], with a sequential consistency model.

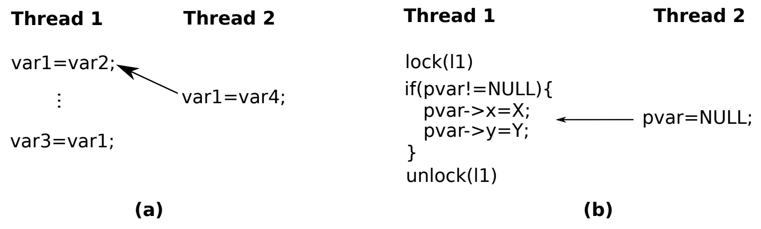


Figure 1.5: Examples of data races. (a) An example of a common data race, and (b) An example of asymmetric data race.

Detecting Data Races

A key type of concurrency bug is a data race. A data race occurs when two or more concurrent accesses to one shared variable (with at least one write) are executed without proper synchronization, and therefore it may result in an undesired behavior. Figure 1.5(a) shows an example of a data race; Thread 1 uses var1 to save the value of var2 and then assigning it to var3 in an atomic way, but the atomicity of this action is broken by Thread 2. However, in practice some of these races are not harmful and are intentionally introduced by the programmer in their code for optimization reasons.

There are many approaches in the literature for hardware data race detection [113] [132] [94] and software data race detection [151] [48] [49] [92] [172].

Detecting Asymmetric Data Races

One type of typically harmful data race are the asymmetric data races, which may occur when some threads are properly synchronized and others are not. In these races, there is a well-tested, correct thread that accesses shared variables with appropriate synchronization. In addition, there is a second thread, typically external to the well-tested application, which is insufficiently tested and accesses shared variables without correct synchronization protection. These threads are called the *safe* and the *unsafe* thread, respectively. An asymmetric race occurs when the *safe* thread is executing a critical section protected by synchronization and the *unsafe* thread corrupts the state of the critical section or reads inconsistent data. In these cases, the program can lead to unexpected results.

Figure 1.5(b) shows an example of an asymmetric data race, where Thread 1 protects the access to the variable *pvar* inside a critical section, and Thread 2 modifies *pvar* without adquiring the lock.

These races are common in bug reports, and can often appear in well-tested codes that interact with third-party or legacy routines [140]. They are likely to be harmful because the data being corrupted is critical data already protected by synchronization.

Asymmetric data races are very common in software development projects [140]. Microsoft lists two main reasons for them [140] [137]. The first one is that code developed by good software developers has to share synchronization operations with code developed outside, in third party libraries. The latter can be racy and buggy. The second reason has to do with legacy code. For instance, a library may have been written assuming a single-threaded environment, but later the requirements change to a multithreaded environment. This requires that all users acquire and release the appropriate synchronization at the appropriate times.

There are some software implementations which target asymmetric data races [137] [140]. This thesis contributes to the first hardware solution to detect and tolerate asymmetric data races in production runs (Chapter 4).

Detecting Deadlock Bugs

Even having all the critical sections synchronized, several problems can arise, like deadlocks (see Section 1.3.1). There are several approaches to handle deadlocks such as those proposed in [22] [48] [151].

Detecting Atomicity Violations

Atomic violations happen when programmers fail to enclose memory access that should execute atomically, and that can lead to incorrect behavior because of the conflicting access from other interleaving threads.

There are some architectural proposals that address the problem in different ways. One example of a hardware detection of atomicity violations is Atom-aid [94]. The main idea of this implementation is to group the instructions in chunks, intelligently formed, to hide and reduce the number of potential atomicity violations. Another example is AVIO [93], which runs several training executions to find invariant interleaving patterns, which will be used to check the correct execution of future runs.

Deterministic Replay

Deterministic replay is a technique to re-execute a non deterministic program in a deterministic way. To achieve this goal, when the program runs, some critical information is stored to reconstruct exactly the same execution.

Therefore, a deterministic replay system has two modes:

– Record mode: records the logical thread information (data interleaving, schedule information, etc), that is different depending on the implementation.

– Replay mode: reproduces the execution behavior of the program by enforcing the recorded information.

For a practical implementation, the system should record almost at production-run speed, it should keep their logging requirements as low as possible and it should replay at a speed similar to the initial execution. These requirements are very difficult to meet without hardware support, and because of that, hardware approaches [107] [10] [114] [115] [181] are more common than software approaches [128] [34].

Other Techniques to Simplify Parallel Debugging

Another set of approaches to simplify parallel programming are the support of more intuitive memory consistence models or forcing the system to be deterministic. BulkSC [29] implements sequential consistency (See Section 1.1.3) in a multicore processor without sacrificing performance. RCDC [44] is an example of a deterministic multiprocessor architecture with a hybrid hardware-software approach.

1.4 Transactional Memory (TM)

Transactional memory (TM) [72] is one of the abstractions that has become very popular in the last years in academia and industry, and it has resulted in several important projects in software (like TM support for gcc [52]) as well as in hardware (Intel Haswell [80], IBM Blue Gene/Q [66] [175], IBM System Z [77] or Vega 2 [39]).

TM is a concurrency control mechanism (analogous to database transactions) that simplifies parallel programming and avoids many of the problems associated with the use of locks [173]. With TM, programmers enclose the piece of code that they want to make atomic in a transaction, which has the properties of atomicity (it executes completely or it is not executed at all) and isolation (other transactions or operations cannot read or write data that has been written by the transaction currently in progress). These properties are automatically supported by the underlying TM system (runtime and hardware support). Usually transactions are explicitly delimited by the programmer through a software constructor, but it can also be implicitly delimited though lower-level operations [62].

TM is a coarse grain synchronization mechanism, which is easier to program and less prone to errors than locks, but thanks to the speculation, its performance is comparable or better than fine grain locks (depending on the TM implementation). Programming with transactions has the big advantage of composability, that is the capacity of composing a large transaction from several smaller transactions together allowing the handling of nested transactions. This is not possible with locks.

TM can be implemented both in hardware or software. Hardware implementations (HTM) achieve better performance, in many cases far better than fine grain locks, but their weak point is the flexibility and the hardware cost. Software implementations (STM) are totally flexible, but in many cases the overhead produces an intolerable performance decrease. To balance the inconveniences and advantages of both alternatives, some hybrid implementations (HyTM) have been developed [84] [106].

TM was seen by some researchers as a research toy [27], but it became a serious tool from the moment when the manufacturers decided to turn TM into a reality in their new generation of microprocessors. Recently, IBM, Intel, Sun, AMD or Azul systems have announced multicore processors implementing some TM support [66] [80] [168] [6] [39]. In software, there are also very popular alternatives, like the support of TM by gcc [52], DSTM2 [121] by Oracle, the Intel compiler [158] or Microsoft's NET framework [105].

1.4.1 Basic Concepts

TM executes atomic blocks of instructions (transactions) speculatively and in parallel [67] [87]. For a transaction to be executed successfully, it has to complete in isolation with respect to other transactions. To achieve that, the basic requirement is that no other transaction reads or writes data that the local transaction modified, and that no other transaction writes data that the local transaction reads. The set of addresses read by a transaction is the **read set** and the set of addresses written by a transaction is the **write set**. A transaction **commits** when finished correctly and the changes are made permanent. Furthermore, a transaction may **abort** at any moment (for example, because of data conflicts), causing all of its prior changes to be rolled back.

Conflict Detection and Resolution

The concurrency control mechanism is composed by a conflict detector and a conflict resolution mechanism. To detect conflicting access to the same data among transactions, a conflict detection mechanism is needed. A conflict detector detects when a transaction is accessing to data that another concurrent transaction is modifying, or when it is writing data that another concurrent transaction has read. There are two basic classes of conflict detectors based on when the conflict is detected:

– **Eager conflict detection** - it is active, and the conflicts are detected on the fly.

– **Lazy conflict detection** - it is passive, and the conflicts are detected at the end of the transaction.

Depending on the implementation, the benchmark or the input, one policy will behave better than the other. The eager policy is used in **pessimistic concurrency control** (the conflict detection and resolution occur when the conflict is produced), and it can save a lot of work of doomed transactions. However, lazy policy is used in **optimistic concurrency control** (conflict detection and resolution can happen after the conflict is produced), which allows multiple transactions to access data concurrently and to continue running even if they conflict, as long as the TM detects and resolves these conflicts before a transaction commits. This provides considerable implementation leeway (for instance, conflicts can be resolved by aborting a transaction or by delaying one of the conflicting transactions).

Figure 1.6 shows an example of how both policies behave in two different scenarios: in (a) the write transaction (T2) commits earlier than the read transaction (T1) (and therefore, from the point of view of the system, T2 is executed before T1), and in (b) just the opposite happens (T1 commits before T2). In (a) the conflict is detected correctly with both strategies, because T2 breaks the atomicity of T1 by writing the A value. However, in (b) the atomicity is not broken because T2 commits after T1, and the B value read by T1 is correct (it is the old value). In this case, the eager conflict strategy will detect a conflict that finally does not break the atomicity, and the lazy strategy does not detect the conflict because it check them in the commit phase.

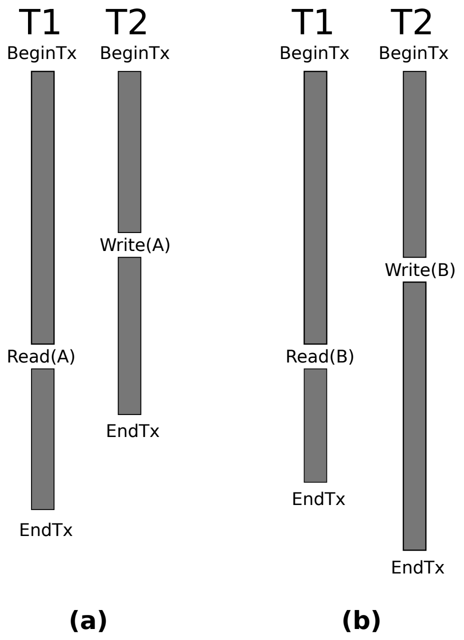


Figure 1.6: In (a), both eager and lazy conflict detection would detect a conflict. In (b) only eager conflict detection detects a conflict.

Depending on the implementation, TM can detect conflicts with different levels of granularity. For example, conflicts can be detected at object level in some software implementations, or at cache line level granularity in most of the hardware implementations. After the conflict is detected, the resolution mechanism decides what to do with the implied transactions (continue, abort, stall, etc.). Furthermore, a contention manager may be required to execute the conflict resolution policies and to avoid some problems that can arise, such as deadlock or livelock.

Isolation

Related to conflict detection, a TM system can also be classified in terms of the level of isolation of the transactions [101]:

- **Strong isolation** - the data enclosed in one transaction is protected against conflicts with transactional and non transactional data. This approach has a significant overhead, so that it is implemented mainly in HTM systems.
- **Weak isolation** - the data enclosed in one transaction is only protected against conflicts with transactional data. This option is more common in STM systems due to its low overhead.

Version Management

To resolve conflicts, the TM system generally needs to abort and rollback one or more transactions. For doing so, it needs to maintain two versions of data during the time the transaction executes, just in case the transaction rolls-back. The **data version management** is in charge of keeping the new and the old data, and depending on where the version of data is saved, we can classify them as:

- **Eager version management** - this approach maintains the new speculative data in place at memory, whereas the old data is maintained in a separate virtual cacheable memory.
- **Lazy version management** - this approach maintains the old data in place, and the speculative data is saved in another location (a log, a specific cache for speculative writes, etc.).

Eager version manager makes the commits fast, and the aborts slow, whereas lazy version manager makes the commits slow, and the aborts fast. If one transaction completes without conflicts, it commits by making the speculative written data accessed by the transaction visible to the rest of the system, and the old data is discarded. In case the transaction is aborted because of conflicts with other transactions, the speculative data is discarded, and the old data is restored.

Nested Transactions

Concerning nesting transactions, a TM system can handle them in three ways:

- **Flattened nested transactions**: All the transaction are flattened and treated as only one (the outer transaction).

– **Closed nested transactions**: A closed transaction can abort without aborting the outer transaction, and when it commits, the results are seen by the outer transaction (but not for the rest of the system).

– **Open nested transactions**: When an open transaction commits, its changes are seen by the outer transaction and also by the whole system. On aborts, open transactions have the same behavior as closed transactions.

These basic mechanisms are enough to implement a complete TM system. Next we will review some representative hardware, software and hybrid implementations.

1.4.2 Hardware Transactional Memory (HTM)

TM implementations in hardware are the best in terms of performance. Many of them were developed in academia [72] [62] [183], but also industry has developed several HTM approaches in recent years [167] [6] [80] [66] [39], which reflect the potential of TM for the future of parallel programming.

In the following, we describe three academic implementations and five commercial HTM proposals.

The First HTM System: the Herlihy and Moss Implementation

The first HTM implementation was proposed by Herlihy and Moss [72] in 1993. The hardware added is restricted to the first level caches and some new instructions, and TM is implemented by modifying the cache coherence protocol and exploiting the access rights (implemented in most of the cache coherence protocols).

The implementation proposed is based on a snoopy protocol, a separate cache for the speculative data and new transactional cache states. Moreover, the processor maintains a state that indicates if there is or not an active transaction in the processor. The transactional cache behaves as a normal cache if the local core is not executing a transaction.

The basic behavior is the following: the transactional cache holds all the tentative writes, without propagating them to other processors or to main memory unless the transaction commits. If the transaction aborts, the lines holding tentative writes are dropped (invalidated); if the transaction commits, the lines may then be snooped by other processors and written back to memory upon replacement. The transactional cache augments the classical cache states (shared, modified) with some additional tags to handle speculative data and tracks conflicts by slightly modifying the cache coherence protocol (by adding information to distinguish transactional messages). The conflicts are detected by the transactional cache coherence controller, which snoops all the coherence messages to know the possible conflictive remote accesses.

The idea of taking advantage of the cache coherence protocol to detect conflicts of data among transactions would be used later by many other proposals (LogTM[109], LogTM-SE[183], etc.).

Transactional Coherence and Consistency (TCC)

Transactional coherence and consistency (TCC) [62] is a shared memory model based on TM, and one of the most representative lazy HTM systems (lazy conflict detection and lazy version manager).

In this system all the instructions execute inside transactions, which are always the basic unit of work, communication, cache coherence and memory consistency. To develop software for this system, the programmer (or the compiler) has to divide the program into transactions. Optionally, the programmer can also specify order among transactions, which allows speculative parallelization of sequential programs.

In TCC, the write buffer stores all the updates until they commit or abort. The read bits in the cache maintain the read set, the modified bits in the cache tracks the write set and the rename bits are optional bits to optimize the protocol.

Each transaction produces a set of writes that are committed atomically to shared memory only when the transaction finishes successfully. Once a transaction completes, the system has to arbitrate for the permission to commit its writes. When the permission is granted, the processor broadcasts the writes to all the system.

TCC has a greatly simplified coherence protocol, since it only needs to manage sequencing among entire transactions, and not among individual loads and stores. Also, it is consistent because it imposes sequential order among all the transaction commits. Moreover, it is coherent: stores are kept in a buffer until the end of the transaction (to maintain atomicity), and several processors can keep and modify the same data. At the end of the transaction, the processor notifies to all the other processors about the changes, and makes the proper invalidations and updates to keep the coherence, and at the same time determines if there are data conflicts that force the transaction to abort and restart and reload the correct data.

The main drawback of TCC is the high broadcast bandwidth required to send the commit packages, which include all the modified data of the committed transaction

LogTM-SE: Log-based Transactional Memory - Signature Edition

LogTM [109] is the most representative of the eager systems (eager conflict detection and eager version manager). There is a later version of LogTM called LogTM-SE [183], which includes signatures (see Section 1.5) to manage conflicts. We describe this implementation in detail because it is the basis of one of the contributions of this thesis (Chapter 3).

Signatures, usually composed of a register and one or several hash functions, can keep a probabilistic representation of an unbounded number of addresses in a bounded space, at the cost of false positives (a positive that actually is not). The addresses are inserted in the signature by hash encoding the address and setting the positions on the register that corresponds with the result of the hash function. Checking if an address is in the signature is an analogous operation, but checking the positions instead of setting (if all of them are one, the result of the check is positive). Notice that signatures never report a false negative (a genuine match is always reported).

Beside signatures, LogTM-SE has the characteristic of supporting transactional data overflowing from the local cache without aborting the transaction, because the old versions of speculative data are saved in a software log, and not in the lower levels of the cache hierarchy as in other implementations.

LogTM-SE builds upon a conventional shared memory multiprocessor with two (or more) levels of private caches that keep coherent by a MESI directory protocol [41]. The old values are saved in a per-thread log in cacheable virtual memory, which is allocated on the thread creation. On a store, LogTM-SE appends to the log the virtual address of the stored line and the line's old value. Writing log entries generates less overhead than one might expect. Log writes will often be cache hits, because the log is cacheable, thread private, and most transactions write few lines. To abort, LogTM-SE must undo the transaction by writing old values back to their appropriate virtual addresses from the log.

Figure 1.7 shows the basic hardware organization of LogTM-SE. The read and write sets are tracked with two signatures, which implement insert, check and clear operations (see Section 1.5), and the eager conflict detection is performed using the MESI cache coherence protocol implemented in the system. In LogTM-SE, the MESI protocol is implemented using a directory.

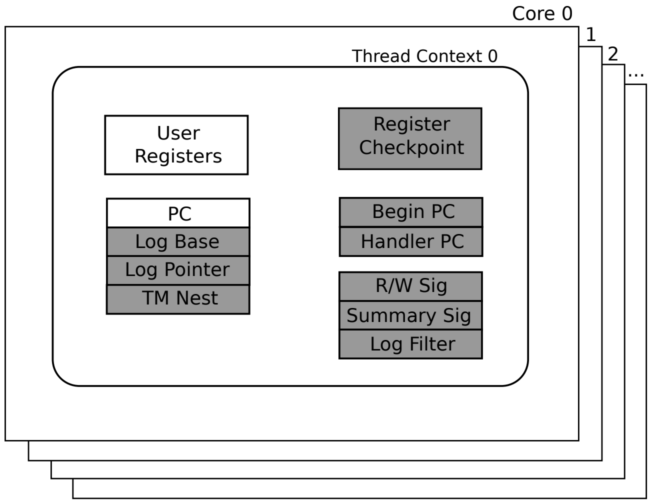


Figure 1.7: LogTM-SE hardware organization. The shaded elements are the LogTM-SE specific state.

In this protocol, each read/write miss generates a request to the directory, and the directory forwards this request if necessary. In case the line is shared by several L1 caches, or owned in exclusivity by one of them, the directory forwards the request to the involved caches. In case the line is not in L2 cache, a L2 miss is produced, and the data is requested to memory.

LogTM-SE performs eager conflict detection in several steps: (a) the requesting processor sends a coherence request to the directory, (b) the directory responds and possibly forwards the request to one or more processors, (c) each responding processor examines some local state to detect a conflict, (d) each responding processors ack (no conflict), or nack (conflict) the request, and (e) the requesting processor resolves any conflict.

To support conflict detection, LogTM-SE introduces some changes on the original MESI directory protocol. If a L1 cache data is evicted, the L2 does not update the exclusive pointer or sharer's list (sticky states). This ensures that a subsequent request will still be forwarded to the evicted L1 line, allowing the conflict detection.

If the L2 replaces transactional data, it loses the corresponding directory information, as the main memory does not maintain directory information. As a result of the inclusion property, subsequent requests to the same data result in a L2 miss. But to preserve correctness, the L2 conservatively broadcasts the coherence request to the L1s, allowing them to check their signatures. To avoid multiple broadcasts for the same line, the L2 rebuilds the directory state by recording the L1s' responses. If an L1 NACKs the request due to a conflict, the L2 directory goes to a new state that requires L1 signature checks for all subsequent requests. A line leaves this state when the request finally succeeds.

For conflict resolution, the contention manager is activated and it executes a timestamp resolution policy, so that if the requester transaction is younger, it should be stalled until the older transaction finishes (commits or aborts), but if the requester is older, the younger transaction aborts. Furthermore, LogTM-SE does not allow us to cache a line in the L1 cache that is in the write set of another core, which ensures isolation.

LogTM-SE is also able to operate in multi-threaded cores, adding additional mechanisms to detect conflicts among threads in the same core. Each thread context maintains its own read and write signatures.

Regarding the version management, the eager approach uses a software log allocated in thread-private memory. LogTM-SE uses an array of recently logged lines for each thread context as a simple but effective log filter. When a thread stores to a line not found in its log filter, LogTM-SE logs the line and adds its address to the log filter. Stores to addresses in the log filter are not logged.

Commits in LogTM-SE are a local and fast operation, which consists of clearing the local signatures and resetting the log pointer. Aborts are managed using a software handler, which walks the log in FIFO order restoring transactional modified lines, and after that, the read and write signatures are cleared.

With a naive directory protocol, cache victimization could lead LogTM-SE to miss some signature checks and hence miss some conflicts. LogTM-SE avoids this case by extending the directory protocol to use LogTM's sticky states [109]. LogTM-SE's caches silently replace lines in states E and S and write back lines in state M. When evicting a cache line, however, LogTM-SE does not change the directory state, so that the directory continues to forward conflicting requests to the evicting core. Thus, LogTM-SE allows transactions to overflow the cache without a loss in performance.

Sun Rock TM Implementation

The Sun Rock [167] [168] [45] [32] was the first commercial HTM implementation. It has very modest TM support (best effort approach), and can be used for lock elision or for some hybrid implementations. Unfortunately, the Sun Rock was canceled in 2009, but it showed the way to follow for other manufacturers.

Rock uses two new instructions to support TM, one that denotes the beginning of a transaction and the other that denotes the end. Rock also adds a s-bit in cache lines. The transactional loads set the s-bit in the corresponding cache memory line, and if a cache line with its s-bit set is evicted, the transaction is aborted.

Stores within a transaction are placed in the store queue in program order. The addresses of stores are sent to the L2 cache, which then tracks conflicts with loads or stores from other threads. If the L2 cache detects such a conflict, it reports the conflict to the core, which then aborts the transaction. When the commit instruction executes, the L2 locks all lines being written by the transaction. Locked lines cannot be read or written by any other threads. This is the point at which other threads view the transaction's loads and stores as being performed, thus guaranteeing atomicity. The stores then drain from the store queue and update the lines, with the last store to each line releasing the lock on that line. The support for locking lines stored by a committed transaction is the primary hardware mechanism added to Rock to implement TM.

Unlike many other HTM systems, the Rock processor implementation ensures weak isolation (detection of conflicts only among transactions).

AMD ASF (Advanced Synchronization Facility)

The Advanced Synchronization Facility (ASF) [6] [37] is an AMD64 extension to provide a very limited form of HTM support. It exposes a mechanism for atomically updating multiple independent memory locations, and allowing software to implement the intended synchronization semantics. ASF is a high level specification, and it does not provide any specific hardware implementation.

ASF specify the execution of the atomic sections of code in a speculative way, and if a conflict is produced, ASF report it to the software, which can retry the transaction as desired.

Furthermore, despite the fact that ASF protects memory at cache-line granularity, software can work on the level of memory objects because:

– ASF-protected memory objects have a size of up to 64 bytes and are naturally aligned (all ASF implementations should have cache lines of at least 64 bytes).

– The speculative region does not reference more than four objects (this is the minimum guarantee, but more may be supported depending on the architecture).

– Memory objects protected using ASF do not share cache lines with memory objects that are not protected. (False sharing may lead to unwanted protection, exceptions and unnecessary aborts).

Some limitations are that ASF supports only a limited form of nested speculative regions, and that only operates on cacheable data and has a weakened memory-access-ordering model in certain aspects.

Chung et all [38] proposed an ASF hardware design to implement in a future AMD out-of-order processor, which is close to the classical approaches on cache-based HTM designs [183] [62].

Intel Haswell TM Implementation

The new Intel Haswell architecture [80] includes some synchronization extensions to take advantage of the underlying TM system. The Intel's Transactional Synchronization extension (TSX) describes two software interfaces for HTM in Haswell, one is for Hardware Lock Elision (HLE) [138], and the second mode is Restricted Transactional Memory (RTM), which is similar to classical TM proposals.

The HTM implementations following the specifications of TSX have cache line granularity and strong isolation. Typically, conflicts cause the transaction to abort, and false conflicts can occur because of cache-line granularity. TSX also supports nested transactions, which are managed by flattening the nested transactions in a single transaction. Transactions can only be used with write-back cacheable memory operations, and not all the instructions can be used safely inside a transaction (for example instructions related with interrupts, I/O, virtualization, etc). There are also limits to the size of the transaction (probably because the size of the transaction is restricted to the L1 cache).

The first of the interfaces defined is **RTM**, which exposes nested transactional memory to the programmer. For implementing RTM, there are three new instructions, one for starting the transaction, one to indicate the end of a transaction and a explicit instruction to trigger an abort.

The other interface defined is **HLE**, which introduces two new instructions to denote the bounds of the lock elision (see Section 1.4.5). One instruction indicates the beginning of a region for lock elision, and the other instruction is for releasing the lock address when the HLE region finishes. The HLE region is treated as a transaction, and the memory address of the lock instruction is added to the read set (but the lock is not acquired). If a conflict occurs, the transaction is aborted, and the HLE region is executed again, but this time acquiring the locks (without HLE hardware support).

Regarding the **architectural support**, unfortunately Intel has not revealed many details about the architecture, but we can outline some ideas suggested by D. Kanter [80] and that probably match the actual architecture. The Haswell coherency changes are probably restricted to L1 and L2 cache (in a three level cache system). Probably a read and a write bit are used per cache line and thread to indicate that the line belongs to the read or the write set of the transaction. The L3 cache would store the old data. It is also likely that the size of transactions is restricted to the L1 data cache. The conflicts would be detected eagerly through the existing cache coherency protocol. To commit a transaction, the L1 data cache and L2 cache controllers make sure that any cache line in the write set (WS) is in the Modified state and clean the WS bits. Similarly, any cache line that is in the read set (RS) -but not the WS- must be in the Shared state and the RS bits are cleared. To abort a transaction, the cache controllers change all the WS lines to the Invalid state and clear the WS bits and the RS bits.

IBM Blue Gene/Q TM Implementation

The last generation of the IBM Blue Gene [66] [175] is a 18-core high-performance energy-efficient computing system that also incorporates HTM.

The main TM characteristic of the Blue Gene is the multiversioned L2 cache used to support speculative execution, TM and atomic operations. During memory speculation, the L2 cache tracks state changes caused by speculative threads and keeps them separate from the main memory state. The speculative data is only visible for the thread that writes it. At the end of the speculative code, the changes can either be made permanent (commit), or be reverted (abort). Also, the L2 track for Read-after-Write (RAW), Write-after-Write (WAW) and Write-after-Read (WAR) conflicts. The L2 can be configured to react to a conflict with an invalidation, with a notification to the software, or both. In the second case, the software decides which transaction to abort to resolve the conflict.

IBM System Z TM Implementation

The last generation of the IBM system z CPU [77] also implements a pure HTM system and incorporates architectural features to support debugging and testing. It introduces six new transactional instructions: TBEGIN (it indicates the beginning of the transaction), TBEGINC (it mostly behaves like TBEGIN), TEND (it indicates the end of the transaction), ETND (it is used to load the current nesting depth into a general register), NTSTG (non-transactional store; unlike a normal store, it is committed to memory even in the case of transaction abort, mainly for debugging purposes) and TABORT (it causes an immediate abort). The main implementation components of the TM system are a register file to save the old versions of transactional data, a cache directory to track the cache lines accessed during the transaction, a store cache to buffer transactional stores until the transaction ends, and firmware routines to perform various complex functions. Furthermore, it includes other architectural registers to support and track different events of the transactional behavior.

1.4.3 Software Transactional Memory (STM)

The main advantage of STM over HTM is the flexibility to implement different strategies of conflict detection and version management, as well as the capacity to manage unbounded transactions. Also, STM is easier to modify and evolve than HTM, and it can be integrated easily with the existing software systems. The main drawback of STM is its large runtime overhead.

The STM precursor scheme was proposed by Lomet in 1977 [91], who proposes a programming language construct very similar to STM, but with a different name. Lomet analyze the disadvantages of synchronization mechanisms (locks, semaphores. monitors, etc), and noted that programmers use these mechanisms to execute sets of code atomically. However, the term STM first appeared in a paper of Shavit and Touitou [153] in 1995, which is considered the first STM implementation. In this first implementation the programmer had to declare which locations might be accessed by the transactions and to propose the memory updates in advance. This approach inspired many early non-blocking STM implementations.

There are some basic core techniques that are used across most of the STM systems: concurrency control metadata, version management and read and write track. To associate the **metadata** (data that describes the data) with the locations that the program is accessing, there are two basic approaches: an object-based STM approach (held with each object) or a word-based STM approach (associated with each memory location).

Regarding **version management**, STM needs and undo-log for eager version management (to save the old values that would be restored if an abort occurs), and a redo-log for lazy version management (with the values that will be written to memory if the transaction commits).

There are two types of **concurrency control**. Pessimistic concurrency control needs a mechanism to track read and write sets, so that the transaction can release any lock that has acquired. In optimistic concurrency control, it is the transaction which detects the conflicts.

Beyond these core techniques, we can classify the STM systems in four groups, depending on the specific implementation: Lock-based STM with Local version numbers, Lock-based STM with Global Clock, Lock-based STM with Global Metadata and Nonblocking STM Systems. Below, we briefly describe each one of these groups:

Lock-based STM with Local Version Numbers

This variant combines a pessimistic concurrency control for writes (using locks acquired dynamically) with optimistic concurrency control for reads, implemented by checking version numbers during validation, which are incremented independently in each piece of STM metadata. The main algorithmic choices are eager or lazy conflict detection, and they lock the locations when they are accessed (encounter-time locking or ETL) or in the commit phase (commit-time locking or CTL). ETL supports both eager or lazy version manager, detecting conflicts among running transactions (whether or not they commit). CTL only support lazy version management, which allows supporting lazy conflict detection. This approach has been used in many STM systems [1] [2] [68] [145] [146].

Lock-based STM with Global Clock

Unlike local version numbers, this implementation uses a global clock to maintain the version numbers, which is incremented globally in the process. These implementations can easily provide opacity (the property of guaranteeing that a transaction always sees a consistent view of memory as it runs). One good example of this approach is TL2 [46].

Lock-based STM with Global Metadata

The techniques that use global metadata do not have individual locks or version number associated. The only shared metadata are global structures and they only use a fixed amount of global state for detecting conflicts. Because of this, this approach may have scalability problems. With this approach, the number of atomic operations involving running and committing transactions can be reduced. JudoSTM [118] and NOrec [42] are good examples of this approach.

Nonblocking STM Systems

The nonblocking synchronization implementations are characterized by the absence of blocking when a process tries to access a concurrent object (data object shared by multiple processes). Instead of that, the process can either abort their own atomic operation, or abort the atomic operation of the conflicting process. One of the most influential approaches is DSTM [71], which was the first that didn't need to specify the locations to be accessed by a transaction in advance. Other designs that were strongly influenced by DSTM were also developed in [60] [152] [51] [99] [158].

1.4.4 Hybrid Transactional Memory (HyTM)

HyTM is a compromise between HTM and STM. It is becoming a very interesting approach because it takes the best of both worlds: the flexibility and relatively low cost implementation of STM, and the performance of HTM.

There are two ways of building a HyTM system:

1. A complete HTM system that can manage short transactions. By default, the transactions are executed in hardware, but when the transaction exceeds the hardware resources, the transaction is aborted and re-executed by software.

2. Hardware support to accelerate some specific STM functions, such as, for instance, conflict detection.

The HyTM proposed by Kumar [84] is an example of HyTM that fits in the first class of HyTM systems: if the HTM system can not handle the transaction, the transaction is restarted in an object based STM system.

SigTM [106] proposes a HyTM system with strong isolation guarantees. This implementation fits in the second class of HyTM systems: it uses hardware signatures (Section 1.5) to perform conflict detection among threads and to reduce the overhead of the STM. It is also the first Hybrid implementation that guarantees strong isolation, because the conflicts are detected through the cache coherence protocol, and it is easy to track all read and write data with signatures tracking all the cache coherence messages.

1.4.5 Other Speculative Techniques

There are other speculative techniques based on speculation that are pretty similar in concept to TM.

Thread-Level Speculation (TLS) [63] [157] [160] is an automatic technique to extract parallelism from a thread. Usually the parallelism is extracted by splitting the dynamic execution path of a single-threaded application into multiple ordered threads. These threads are executed concurrently, and only the first thread in order is nonspeculative. When the nonspeculative threads complete, the next thread in order becomes nonspeculative. The speculative threads have to maintain the speculative writes in a special buffer, and only when the thread becomes nonspeculative, are these writes seen by all the system.

Speculative Lock Elision (SLE) [138] executes the region of code inside the lock speculatively. In case of no conflicts, the execution would be very fast, and in case of detecting conflicts, the old data values are restored, and the region of code is re-executed acquiring the corresponding lock. This technique was used in hardware implementations like Haswell [80].

1.5 Background on Signatures

New paradigms and tools related to parallel programming and debugging require hardware support to achieve a reasonable performance. One promising hardware support element for parallel architectures are signatures, which can be used in a variety of different tools with different purposes, and provide a significant performance improvement compared with their cost.

A signature keeps a probabilistic representation of an unbounded number of elements in a bounded space, and it is used to check if an element belongs to the set of elements previously inserted in it. As a consequence of the bounded space, aliasing among elements can be produced, and therefore, the check operation can return false positives (a positive that actually it is not). However, a check operation never results in false negatives (a genuine match is always reported), because usually it is not possible to remove individual elements from the signature.

One of the most critical features to implement in a TM system (Section 1.4) is the conflict detection, so that many approaches have used signatures for this purpose. This is the case of LogTM-SE [183], SigTM [106], FlexTM [155] or DynTM [96]. Another example of the use of signatures is BulkSC [29], a novel multicore architecture that provides sequential consistency with a high performance by executing atomic blocks of code atomically and in isolation, using signatures to detect conflicts and preserve isolation. Signatures are also used in code analysis tools such as SoftSig [169], which exposes hardware signatures to the software for code analysis and optimization. Concerning data race detection, SigRace [113] is a very efficient race detection tool with signatures as a key element. Related also with debugging, there are implementations of deterministic replay with signatures as hardware support, such as DeLorean [107]. Other very popular uses of signatures, apart from applications related to parallel programming, are: web cache [78], packet classification [3], packet inspection [82], network services [180], grid services [89], cloud computing applications [127], routing protocols [125], etc.

These are only some examples of the wide variety of tools that are already using signatures as a key element of their behavior, which indicates that signatures are a good candidate to include in a general purpose processor to support these and future tools. Chapters 5 and 6 presents our contribution on signatures.

For the applications discussed in this thesis the elements inserted and checked on signatures are addresses. Moreover, for high performance we concentrate on hardware signatures.

1.5.1 Fundamentals

A signature is composed of a storage element (usually a register) and at least one hash function. To insert an address in the signature, some bits of the storage elements are set (the bit position corresponding to the result of hash encoding the element). For testing if an address was inserted in the signature, all the bits of the storage element corresponding to hash encode the address have to be set to one. The third basic operation is to clean the signature, that resets all the bits of the storage element.

The rate of false positives is the major figure of merit of a signature, and depends basically on the size of the signature storage element, the quality of the hash functions and on the number and the distribution of elements inserted.

There are many different designs and implementations for signatures (the most popular implementations use Bloom filters [20]), but all of them share two common components: a storage element, and one or more hash functions. Below, we describe these elements.

Hash Functions

The mission of a hash function is to map an address of l bits into a bit position of a $m = 2^c$ bit register. The desirable requirements of a hash function are an easy and fast implementation and an appropriate design to minimize the false positives (good statistical properties).

The simplest choice for a hash function is the **bit-selection**, where each hash value (that points to a bit of the signature register) comes from a fixed subset of the bits of the address. It uses trivial hardware, but it presents limited variation to approximate a uniform distribution of the generated hash values.

The H_3 family of hash functions [26] [139] are more popular in hash implementations. These functions are just a reduction implemented with a tree of XOR gates per result bit of the hash function (determination of the parity of the groups of bits of the input address). It uses fixed boolean matrices to select address bits. Each one of the k H3 hash functions of a signature can be characterized by a $c \times l$ matrix

$$H = \begin{bmatrix} h_{0,0} & h_{1,0} & \cdots & h_{c-1,0} \\ h_{0,1} & h_{1,1} & \cdots & h_{c-1,1} \\ \vdots & \vdots & & \vdots \\ h_{0,l-1} & h_{1,l-1} & \cdots & h_{c-1,l-1} \end{bmatrix} \tag{1.1}$$

whose coefficient $h_{i,j}$ is 1 if the bit j of the address is an input bit of the XOR tree which computes the bit i of the hash value. The hash output value $y = [y_0 y_1 \cdots y_{c-1}]$ of an l-bit address with binary expression $x = [x_0 x_1 \cdots x_{l-1}]$ is computed as

$$[y_0 y_1 \cdots y_{c-1}] = [x_0 x_1 \cdots x_{l-1}] \times H \tag{1.2}$$

Thus, a generic Bloom filter with k hash functions can be completely characterized by k H3 matrices $H_0, H_1, ..., H_{k-1}$.

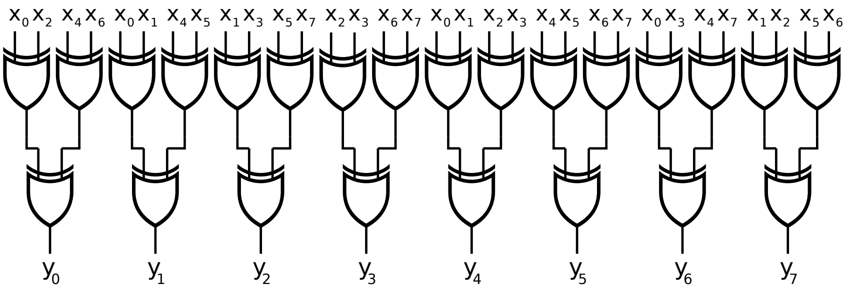

$x_0 x_2$ $x_4 x_6$ $x_0 x_1$ $x_4 x_5$ $x_1 x_3$ $x_5 x_7$ $x_2 x_3$ $x_6 x_7$ $x_0 x_1$ $x_2 x_3$ $x_4 x_5$ $x_6 x_7$ $x_0 x_3$ $x_4 x_7$ $x_1 x_2$ $x_5 x_6$
y_0 y_1 y_2 y_3 y_4 y_5 y_6 y_7

Figure 1.8: Example of H_3 hash function. The inputs are defined by the $H_{example}$ of Equation 1.4.

In general, given fixed boolean matrices, the number of 2-input XOR gates required to implement k hash functions, producing c-bit hash values (that point to a $m = 2^c$ bit register) is given by

$$\#XOR = \sum_{j=0}^{k-1} \sum_{i=0}^{c-1} \left(\sum_{s=0}^{l-1} h_{i,s} - 1 \right) \approx (INbits - 1) \times c \times k \tag{1.3}$$

being $INbits$ the average number of address input bits for generating each output bit $(y_0, y_1, ...)$. For instance, assuming a signature of $m = 1024$ bits ($c = 10$) and one hash function ($k = 1$), with addresses of 32 bits ($l = 32$), and $INbits = 16$ bits, 150 XOR gates would be required. The number of gates can be reduced by applying some optimizations [170].

Figure 1.8 shows a very simple and trivial H_3 hash implementation with one hash function ($k = 1$) for 8-bit addresses ($l = 8$) and a register of $m = 2^8 = 256$ bits, just to illustrate the previous explanation. The input address bits are selected by using the fixed boolean matrix of Equation 1.4 (4 address bits produce each bit of a 8-bit hash value). On average half of the input address bits can affect a given bit of a hash value, and it requires 15 XOR gates.

$$H_{example} = \begin{bmatrix} 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 \end{bmatrix} \tag{1.4}$$

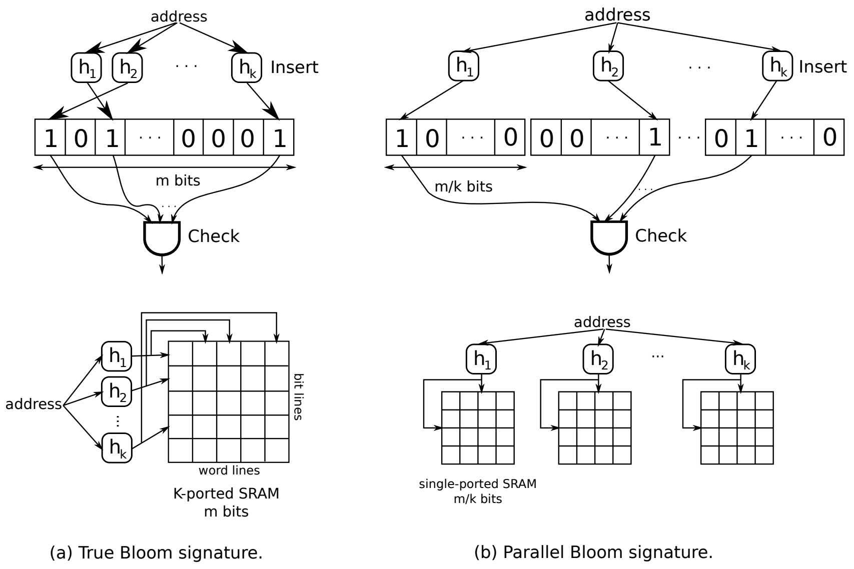


Figure 1.9: Bloom signatures.

Registers

The registers are usually implemented with bit-addressable SRAMs, which can be single or multi ported, depending on the number of hashes for writing the register at the same time. Multiported SRAMS are not area-efficient because the size of a SRAM cell increases quadratically with the number of ports. As described below, multi-ported SRAMs are used in true Bloom signatures, whereas parallel Bloom signatures use single-ported SRAMs.

1.5.2 True Bloom Signature

A true Bloom signature is a signature implemented with a Bloom filter [20]. It is composed by one multi-ported SRAM register of m bits (that is initially set to zero) and k hash functions that return a value in the range $[0..m-1]$, as shown in Figure 1.9 (a).

To insert and address in the signature, the k hash functions are calculated, and the bit indicated by the result of this operations is set to one in the register. The check operation consists of checking if all the bits of the register pointed by the results of the hash functions are all set to one. In case at least one bit is set to zero, the address is not in the signature. If all the bits are set to one, the address was previously inserted (or it is a false positive).

Each hash function in the Bloom filter set or check one bit of the register. All k hash functions are independent, and they map the addresses into k randomly distributed bits of the register.

The most critical design decisions in a true Bloom filter are the size of the register (m) and the number of hash functions (k). Large registers decrease the probability of a false positive, but increase the hardware resources and power/energy required. On the other hand, the probability of false positives depends also on the number of hash functions and the number of elements inserted [147] [20].

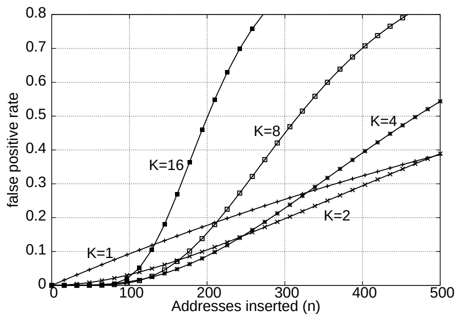
The number of false positives is influenced as well by how the hash functions are implemented (H_3, bit-selection, etc). A theoretical approach for the false positive rate [147] assumes that the hash values are independent and uniformly distributed (very similar to H_3 functions). This leads to the following expression for the lower bound of the false positive rate (P_{FP}):

$$P_{FP} = (1 - e^{-\frac{nk}{m}})^k \tag{1.5}$$

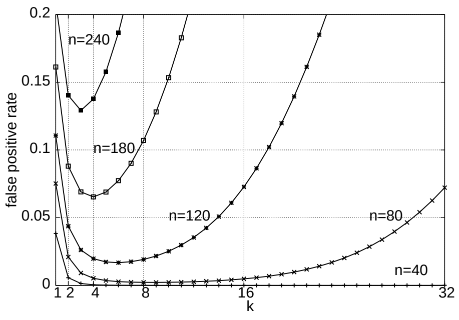
where n is the number of addresses inserted and assuming $m >> 1$. The effect of the number of hash functions is shown in the example of Figure 1.10 (a), where the false positive rate of a signature with $m = 1024$ is represented , with the number of inserted addresses from 0 to 500, and $k = 1, 2, 4, 8, 16$. We see that the best value of k (minimum false positive rate) depends on the number of addresses inserted: with a high number of addresses inserted, the better results are achieved with a small value of k, and with a low number of addresses inserted, the signature requires a higher value of k to minimize the false positive rate. Figure 1.10 (b) shows the evolution of the false positive rate depending on the k value, and for 5 different values of n. We see more clearly in this figure that there is an optimal value of k depending on the number of addresses inserted (n). Specifically, the absolute minimum of the Equation (1.5) is obtained for $k = \ln(2) \times \frac{m}{n}$ [163], and the lower bound of the false positive rate using this value is given by:

$$P_{FP} = 2^{-\ln(2) \times \frac{m}{n}}$$

Since k, in practice can take only integer values, the value of k that minimizes the false positive rate (k_{opt}) is the closest integer to $\ln(2) \times \frac{m}{n}$ that minimizes the value of P_{FP} (Equation



(a) False positive rate vs n default values of k.



(b) False positive rate vs k for default values of n.

Figure 1.10: False positive rate for parallel Bloom filter with $m = 1024$.

(1.5)). For the particular example of Figure 1.10 (b) with $m = 1024$ and $n = 180$, the optimal value is $k_{opt} = 4$.

1.5.3 Parallel Bloom Signature

Figure 1.9 (b) shows an alternative to the true Bloom signature, the parallel Bloom signature [147] [31]. In this case, the m–bit register is split into k registers of m/k bits, each with a hash function associated. In this approach, each hash function operates on one of the m/k–bit registers. Hence, the parallel Bloom signature can be seen as k true Bloom signatures with one hash function and a m/k–bit register (or like k single Bloom filters). The insert operation hashes the addresses and inserts one bit in each m/k–bit register, and the check operation reports that an address is in the signature only if all individual single Bloom filters have the hash mapped bit of their register set to one. The advantage of parallel signatures is that hash functions are simpler and are implemented with fewer resources. Therefore, each of the individual SRAMs of the Bloom filters can be single-ported, thereby reducing significantly the hardware area/power/latency.

The false positive rate depends on the size of the signature m, the number of hash functions k (which is the same as the number of single Bloom filters) and the number of inserted addresses n. Particularly, the theoretical false positive rate is given by the following expression [147] [31]:

$$P_{FP} \approx (1-(1-\frac{k}{m})^n)^k$$

and applying the Taylor series approximation of e^x the equation results in

$$P_{FP} \approx (1 - e^{-\frac{nk}{m}})^k \text{ when } \frac{k}{m} \ll 1$$

This is approximately the same expression obtained for the true Bloom signatures when the size of the signature is much bigger than the number of single Bloom filters ($k/m \ll 1$). Therefore, Figures 1.10 (a) and (b) are also representative of the behavior of parallel Bloom filters.

1.5.4 Other Signature Implementations

There are other interesting signature approaches that improve some aspect of signatures based on Bloom filters, or at least they adapt better to a specific tool. We describe some of them below.

Scalable Bloom Filters

Scalable Bloom Filters [4] are software signatures that can adapt dynamically their size depending on the number of elements. It allows the signature to grow arbitrarily: when a Bloom filter reaches a certain number of addresses inserted (that is correlated with the false positive rate), another Bloom filter is added. This concept of signature requires an arbitrarily number of Bloom filters, and therefore it is more appropriate to implement it in software than in hardware.

AdaptSig [129] or Dynamic Bloom Filters [61] are inspired by this idea, but also both approaches are intended for software implementations.

Cucko Bloom Signatures

Cucko Bloom Signatures [147] represent small read/write sets by adapting Cucko hashing [126]. Cucko Bloom signatures are formed by a hash function that tracks the addresses when their number is low, and eventually is transformed in a Bloom filter when the number of addresses increases. These signatures match the low false positive rates of Bloom signatures with many hash functions when the number of addresses is small, and show the good asymptotic behavior of Bloom signatures with few hash functions when the number of addresses is large.

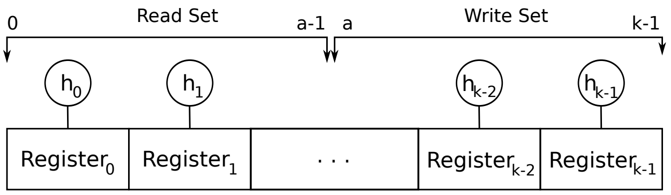


Figure 1.11: High level scheme of the ASYM signature.

Locality-sensitive Signatures

Locality-sensitive signatures [135] are specially designed to exploit locality in transactional memory systems. The design is based on new hash function mappings, so that nearby located addresses are mapped sharing some bits. These signatures are particularly favorable for large transactions that usually exhibit some amount of spatial locality. Their implementation do not require extra hardware.

ASYM Signatures

ASYM signatures [136] are new schemes specially designed for TM systems. Among these signatures, ASYM signatures are of special interest for us because they are related with our *FlexSig* module implementing asymmetric allocation policies. In Chapter 6 we will discuss the differences between both proposals and the advantages of our approach.

ASYM signature deals with the asymmetry of the read and write sets in TM systems. The high level idea is that the ASYM signature configures the number of Bloom filters devoted to each data set. As we show in Figure 1.11, the ASYM signature is composed of k hash functions, each one associated with a register, and a mask register that provides the parameter a which establishes the sizes of the read and write signatures. Specifically, assuming an ASYM signature with k (hash,register)-pairs, and $a \in [1, k-1]$, the hashes $h_0, h_1, ..., h_{a-1}$ are assigned to the read set, and the hashes $h_a, h_{a+1}, ..., h_{k-1}$ are assigned to the write set. The a parameter is dynamically reconfigurable at run-time (it can change among transactions). With a good configuration of the a parameter (it depends on the transaction and the application), the false positive rate of ASYM signatures is improved regarding conventional signatures.

1.6 Conclusions

Parallel programming is the way to continue scaling the performance of programs in multicore processors, but it is not an easy task. Some tools, abstractions and programming languages have been proposed to make this task more accessible for programmers. One of the most popular software abstractions adopted by academia and industry is TM, which simplifies the synchronization of shared memory by defining atomic blocks and ensuring their isolation and atomicity. Furthermore, the development of new tools to support and help debugging (specially related with synchronization bugs) is also important, because with multiple threads accessing shared data, this task has become very complex. Moreover, these new tools and abstractions usually are supported by hardware to achieve a better performance; one common hardware resource used in many tools and applications are hardware signatures, which keep a probabilistic representation of an unbounded number of addresses in a bounded space (low hardware cost), and which can significantly improve the performance of these tools.

CHAPTER 2

EVALUATION METHODOLOGY

Simulation is essential for evaluating new architectures, protocols, or other hardware modifications and additions, because it is more flexible and cheaper than making a prototype in hardware. Furthermore, it is very useful for obtaining information and statistics about performance and events in the system. One of the main advantages of simulation is the flexibility: some simulators allow a very fast execution at the expense of simulation detail, or alternatively, other simulators execute very detailed simulation and obtain a lot of traces and event information (at the cost of simulation time).

In computer architecture, a common way to test the improvement of a contribution is to compare a well known implementation with the new solution. The simulation framework consists in a system simulator and a series of benchmarks running on it.

In this chapter we present the experimental setup used in the evaluation of the different proposals of this thesis. In Section 2.1 we describe the framework used to simulate our proposals, and in Section 2.2 we described the set of benchmarks used.

2.1 Simulator Framework

This section describes the simulators and tools used for this thesis. Section 2.1.1 describes Simics [98] and GEMS [102], the system simulators used for evaluating our CFM-TM proposal in Chapter 3. This framework provides a cycle accurate simulation environment, with support for TM, which we use to obtain detailed statistics of the memory system and of the TM system.

Section 2.1.2 is dedicated to the PIN instrumentation tool [81], which is used for the evaluation of Pacman (Chapter 4) and *FlexSig* (Chapters 5 and 6). PIN has the capacity of enabling the creation of dynamic program analysis tools. With these tools, we build the simulation framework by monitoring the data accessed in transactional and lock based applications, and making the appropriate simulations with the observed data.

Furthermore, in Section 2.1.3 we describe the Rochester STM system [100], which forms part of the experimental framework for the *FlexSig* evaluation. RSTM is an open source STM library, which we use to run transactional code which we monitor with PIN.

Finally, in Section 2.1.4 we describe SESC [124], a cycle accurate simulator used in the evaluation of Pacman.

2.1.1 Simics and GEMS

The ensemble composed of Simics and GEMS is the simulation framework in the evaluation of our TM work in Chapter 3. Figure 2.1 shows an overview of this framework. The main advantages of using Simics and Gems are that they simulate a complete Sparc architecture that can run a complete operating system with a deep level of detail (cycle accurate simulation). Furthermore, with GEMS we obtain a detailed memory system simulator. Despite there being other good alternative simulators, like SESC [124], we choose this environment to simulate for its performance, the strong support of the community of users, and its support for transactional memory.

Virtutech Simics

Simics is a full system functional multiprocessor simulator that tries to maintain the balance between accuracy and performance. It can simulate processors at instruction-set level (with models for Ultrasparc, Alpha, x86, etc), and it can run operating systems, including Solaris, Linux or Windows. Furthermore, Simics allows us to add new user-developed modules to expand the simulator with new features.

In our experiments, we run Simics 2.2.19 with a target system composed of a 16-core UltraSparc-III processor running the Solaris 9 operating system.

Simics provides a deterministic environment for a variety of hardware and software engineering tasks. However, it does not provide micro-architecture timing detail nor cache/memory subsystem timings. To overcome these limitations, GEMS have added these functionalities through a software module for Simics.

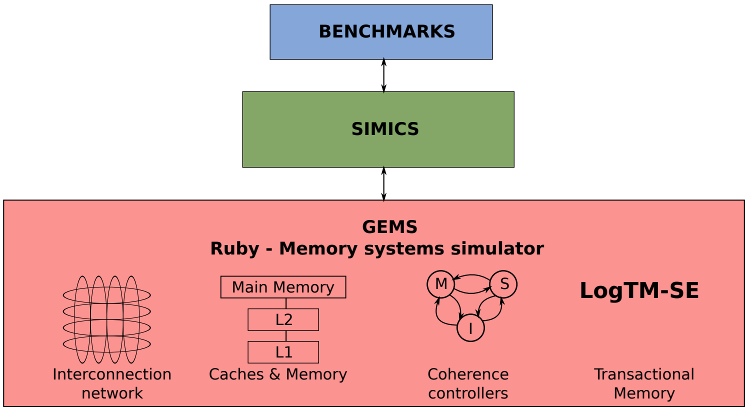


Figure 2.1: SIMICS+GEMS overview.

GEMS

GEMS is a software module developed as part of the Multifacet project at the University of Wisconsin. GEMS is built on top of Simics, enables the simulation of multicore systems, and as shown in Figure 2.1, provides detailed models and timing information of the memory system, coherence controllers and the interconnection network, and supports HTM (specifically LogTM-SE [183] and ROCK [168]).

GEMS is composed by two basic modules: Ruby and Opal. Ruby models the memory hierarchy, using for that a specific language call SLICC, and Opal models the timing of an out-of-order SPARC processor. In this thesis we use only Ruby for simulation, because OPAL does not support TM programs.

Simulated Architecture

The configuration of Simics and GEMS for our experiments is shown in Table 2.1. The simulated multicore processor is composed of 16 single-issue in order cores, with a 32Kb data cache (L1D), and a 32Kb instruction cache (L1I). Both caches are 4-way set associative, with 2-cycles of latency and 64 bytes size lines.

The shared L2 cache has a total size of 8 megabytes, and it is distributed along all the cores of the chip (512 Kb per core). Furthermore, the L2 cache is an 8-way set associative cache with a latency of 15 cycles. The directory, placed in L2, has an access delay of 6 cycles. The

Table 2.1: System configuration.

Cores	16, single issue, in-order
Cache Data Line	64 bytes
L1 I&D caches	32KB, 4-way, 2-cycles latency
L2 cache	8MB, 8-way, 15-cycles latency
Memory	4GB, 500-cycles latency
Directory	6-cycles latency
Network Topology	point to point
Link latency	1-cycle

main memory has 4 Gigabytes and an access latency of 500 cycles. The network topology in our simulation is a point-to-point network (full connected crossbar) with an access latency of 1 cycle.

Moreover, our simulated architecture is configured to support the LogTM-SE HTM [183]. In our CFM-TM proposal we modify this base system to perform our experiments.

Compilation Infrastructure

The compilation infrastructure for the transactional memory benchmarks is a Solaris gcc compiler without modifications. The transactional boundaries are simulated through the Simics "magic instructions", which are special assembly instructions which are functionally no-op in a real machine, but they are reinterpreted by Simics to simulate the TM system.

2.1.2 PIN

PIN is an instrumentation tool used in our experiments with Pacman (Chapter 4) and *FlexSig* (Chapters 5 and 6). It can access information such as register contents, symbols or debugging information.

PIN runs attached to program code, instruments just before it runs (discovers code at runtime), and it does not need to recompile or re-link. With PIN it is possible to set up instrumentation tools (called PinTools), which are composed by two kinds of routines: the instrumentation routines define where the instrumentation is inserted and the analysis routines define the actions to take when the instrumentation is activated. Therefore, a PinTool can

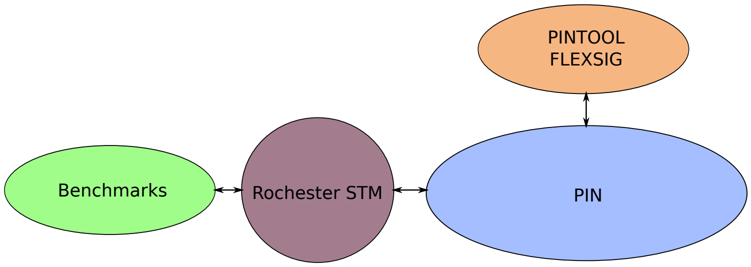


Figure 2.2: Simulation framework of Chapters 5 and 6, composed of our *FlexSig* PinTool for PIN, which instruments the benchmarks running on RSTM.

replace functions in the program by other functions (defined by the PinTool) or examine all the application instructions.

In the particular case of our experiments with Pacman, we used instrumentation routines in PIN to detect the beginning and end of a critical section (when the lock is acquired and released), and analysis routines to monitor the data accesses inside the critical sections (including the code to simulate our hardware module implementation).

In our evaluation of *FlexSig*, we used instrumentation routines in PIN to detect the beginning and the end of the transactions, an analysis routines to track all the transactional accesses and to implement the new signature hardware module. Figure 2.2 shows a high level representation of the simulation framework in this particular case.

The reason to choose PIN for the evaluation of these chapters is because of its flexibility to simulate only the hardware elements which we are interested in, allowing a very good performance and a high level of detail of the simulated modules.

2.1.3 Rochester STM

Rochester Software Transactional Memory (RSTM) [100] is a STM library that can be configured with a wide variety of STM implementations. Specifically, it implements word-based and object based implementations, which are summarized in Table 2.2.

We use RSTM to build a transactional environment and run transactional benchmarks for testing our *FlexSig* work. In our evaluation we use a lazy acquisition and lazy versioning with extendable timestamps [143] to configure RSTM (ET implementation in Table 2.2).

Table 2.2: RSTM implementations.

Granularity	Variant	Implementation	Description
Object based		RSTM	Nonblocking, shallow object cloning, one level of indirection
		Redo-lock	RSTM modified to eliminate the one level of indirection
Word based	Single-Lock	CGL	Coarse Grained Lock, no concurrency
		TML	Transactional Mutex Lock, concurrent read-only transactions
		TML + Lazy	lazy acquire/lazy versioning policy for writers
	Global-Hash	LLT	Lazy-Lazy-Timestamp
		SGLA	Extends the ET with the start time linearization
		ET	Extendable-Timestamps
		Fair	Extends ET with support for transactional priorities
		Strict	Extends ET with support for the acquire and release fences required by the SSS transactional memory model
		Flow	Extends LLT with implicit privatization safety
	Others	RingSW	Single-writer variant of RingSTM [159]
		Precise	Extends TML + Lazy with value-based conflict detection

We choose RSTM over other STM implementations because it is very flexible and configurable, it is well-known and has good support.

2.1.4 SESC

The SuperESCalar (SESC) [124] is a microprocessor architectural simulator written in C++, which models different processor architectures, such as single processors, chip multicore and processors-in-memory. It models a full out-of-order pipeline with branch prediction, caches, buses, and every other component of a modern processor necessary for accurate simulation.

SESC is an event-driven simulator. It has an emulator built from MINT [171], an old project that emulates a MIPS processor. Many functions in the core of the simulator are called every processor cycle. But many others are called only as needed, using events.

SESC is used in the evaluation of Pacman to simulate a chip multiprocessor with a private cache hierarchy system. We use SESC because it is the simulator of the laboratory where the work was carried out.

2.2 Benchmarks

In this thesis we use several sets of benchmarks to evaluate our proposals. SPLASH-2 benchmarks [178] are used in the evaluation of CFM-TM and Pacman, STAMP bechmarks [25] are used to evaluate CFM-TM and *FlexSig*, PARSEC benchmarks [16] are used to evaluate Pacman and "EigenBench" is used to evaluate asymmetric *FlexSig* (in Chapter 6). Table 2.3 shows the list of all benchmarks, and below we describe each benchmark suit.

2.2.1 SPLASH-2

The SPLASH-2 benchmarks [178] are a set of applications and kernels developed to study shared memory multiprocessors, and represent a wide range of computations in the scientific, engineering and graphics domain. The applications are "Barnes", "FMM", "Ocean", "Radiosity", "Raytrace", "Volrend" and "Water", and the kernels are "LU", "Cholesky", "FFT" and "Radix".

We use all SPLASH-2 benchmarks for evaluation of Pacman, and one application of the suite ("Barnes") in the evaluation of CFM-TM. All of them are written in C language, using pthreads, and the critical sections are protected with locks in case of Pacman, and with transactions in the evaluation of CFM-TM.

Table 2.3: Benchmarks.

Suite	Benchmark	Description
SPLASH-2	Barnes	Interaction of a system of bodies
	FMM	Interaction of a system of bodies
	Ocean	Large-scale ocean movements
	Radiosity	Equilibrium distribution of light in a scene
	Raytrace	Real-time raytracing
	Volrend	Rending of a 3D volume
	Water	Water molecules simulator
	LU	Matrix factorization
	Cholesky	Matrix factorization
	Radix	Sorting algorithm
	FFT	Discrete fourier transform
STAMP	Intruder	Network intrusion detection
	Labyrinth	Maze routing
	Vacation	Travel reservation system
	Genome	Gene sequencing
	kmeans	K-means clustering
	ssca	Graph kernels
	yada	Delaunay mesh refinement
	bayes	Bayesian network learning
PARSEC	Blackscholes	Option pricing
	Bodytrack	Body tracking of a person
	Facesim	Simulates the motions of a human face
	Ferret	Content similarity search server
	Fluidanimate	Fluid dynamics
	Raytrace	Real-time raytracing
	Swaptions	Pricing of a portfolio of swaptions
	Vips	Image processing
	x264	H.264 video encoding
	Canneal	Simulated cache-aware annealing
	Dedup	Next-generation compression with data deduplication
	Streamcluster	Online clustering of an input stream
-	EigenBench	Synthetic benchmark for TM systems
Other	Apache	Web Server
	Sphinx3	Speech recognition

Next, we briefly describe all of them:

- **Barnes**: This application is a 3-D system of bodies using the Barnes-Hut hierarchical N-body method. The computational domain is an octree with leaves which contain information on each body, and each internal node of the tree represents a cell (a collection of bodies in close physical proximity). The main data structures are two arrays, one for the bodies and one for the cells. The main phases of the algorithm are: constructing the tree, partitioning the bodies among processors, computing the forces in the bodies (one computation per thread on its own bodies), and updating the position and forces of the bodys (also per thread).

- **FMM**: This application simulates a system of bodies similar to the Barnes-Hut algorithm. However, the simulation is in two dimensions using a different hierarchical N-body method called the adaptive Fast Multipole Method [58].

- **Ocean**: This application simulates ocean movements based on eddy and boundary currents. This grid-based simulation goes through many time-steps until the system achieves a mutual balance. The work done in each step involves configuring and solving a set of partial differential equations on two dimensional grids (with the same resolution in both dimensions) representing the ocean.

- **Radiosity**: This application computes the equilibrium distribution of light in a scene using the iterative hierarchical diffuse radiosity method [64]. The scene is modeled with a large input of polygons, and the benchmark computes the light transport iterations among these polygons. At the same time, the polygons are divided into patches to improve accuracy. In each step, the algorithm iterates over the current interaction lists of patches, subdivides patches recursively, and modifies interaction lists as necessary. At the end of each step we check if the algorithm have converged. The structure of the computation and the access patterns to data structures are highly irregular. Parallelism is managed by distributed task queues, one per processor, with task stealing for load balancing.

- **Raytrace**: This application renders a three-dimensional scene using ray tracing. The scene is represented with a hierarchical uniform grid. A ray is traced through each pixel in the image plane, and reflects in unpredictable ways off the objects it strikes. Each contact generates multiple rays, and the recursion results in a ray tree per pixel.

The image plane is partitioned among threads in contiguous blocks of pixel groups, and distributed task queues are used with task stealing. This application is also included in PARSEC benchmarks.

- **Volrend**: This application renders a three-dimensional volume using a ray casting technique. The volume is represented as a cube of volume elements, and an octree data structure is used to traverse the volume quickly. The program renders several frames from changing viewpoints. A ray is shot through each pixel in every frame, but rays do not reflect. Instead, rays are sampled along their linear paths using interpolation to compute a colour for the corresponding pixel. The partitioning and task queues are similar to those in "Raytrace".

- **Water**: This application simulates the forces and potentials in a system of water molecules. There are two variants of this benchmark: Water-Nsquared and water-spatial. In water-Nsquared, the forces and potentials are calculated using a $O(n^2)$ algorithm, and the integration of the water particles motion over time is made through a predictor-corrector method. The water-spatial variant is a more efficient algorithm for a large number of molecules: the problem domain is represented in a 3-D grids of cells. The advantage of this model is that the thread that owns a cell only has to look at its cell and the neighbours. The movement into and out of the cells causes the communication among threads.

- **LU**: This kernel factorizes a matrix as the product of a lower triangular and an upper triangular matrix (LU decomposition). The division of the work among processes is done through a 2D-block scheme, by which the matrix is divided into square blocks along both axes, and blocks are assigned to the processes in an interleaved manner. Each process is responsible for allocating and working on equal numbers of blocks, with the aim of reducing communication.

- **Cholesky**: This kernel factors a sparse matrix into the product of a lower triangular matrix and its transpose. It is similar to the "LU" factorization kernel, but more efficient. The main differences are that "Cholesky" requires more communication and it is not globally synchronized between steps.

- **Radix**: This kernel is a non-comparative integer sorting algorithm implemented with the method described in [19]. The algorithm is iterative, performing one iteration per radix r digic of the keys. In each iteration, each thread generates a local histogram by passing over its assigned keys. The local histograms are then accumulated into a global histogram. Finally, each thread uses the global histogram to permute its keys into a new array for the next iteration. This last step requires all-to-all communication.

- **FFT**: This kernel computes the discrete Fourier transform and its inverse by a complex 1-D version of the radix-$\sqrt{n}$ six-step FFT algorithm described in [12], which is optimized to minimize communications. The input data set is composed of n complex data points to be transformed, and another n complex data points referred to as the *roots of unity*. Both data sets are organized in $\sqrt{n} \times \sqrt{n}$ matrices, which are divided into sets of rows, each one assigned to a thread that allocates them in its local memory. Communication occurs in three of the six steps, which requires all-to-all thread communication.

2.2.2 STAMP

Stanford Transactional Applications for Multi-Processing (STAMP) is a benchmark suite to evaluate TM systems. We use three STAMP [25] benchmarks ("Vacation", "Intruder" and "Labyrinth") in the evaluation of CFM-TM, and all of them in the evaluation of *FlexSig*.

All of the STAMP benchmarks are written in C, and the critical sections are enclosed in transactions. With CFM-TM, transactions are implemented with magic instructions and in *FlexSig* the transactions uses the RSTM interface.

- **Intruder**: This benchmark scans network packets for matches against a known set of intrusion signatures. The packets are processed in parallel in three phases: capture, reassembly and detection. The capture phase is structured using a FIFO queue, and is enclosed in a transaction. The reassembly phase uses a dictionary that contains a list of packages that belongs to the same session, and it is also enclosed in a transaction. The detection phase is not enclosed in a transaction, so, in general this benchmark has a moderate amount of total transactional execution time.

- **Labyrinth**: This benchmark implements a maze with a data structure representing a three dimensional grid. Each thread sets one start point and one destination point, and a conflict occurs when two threads overlap their paths. To reduce the conflict probability, this benchmark implements the privatization technique described in [176], which consists of each thread making its path with a private copy of the grid, and at the end, when the thread wants to add the new path to the grid, making the validation. If the validation fails, the transaction aborts, and it starts again with an updated copy of the grid.

– **Vacation**: This benchmark emulates a travel reservation system, implemented as a set of trees that keep track of customers and their reservations. The execution of the benchmark consists of several threads (clients) interacting with the travel system database in three different ways: reservations, cancellations and updates. Each one of these interactions is enclosed in one transaction, and consequently, "Vacation" spends a lot of time on transactions. These transactions are medium length with moderate read and write set sizes, and they have low to moderate levels of contention (depending on the input).

– **Genome**: This benchmark takes a large number of DNA segments and matches them to reconstruct the original source genome. The process is divided in two phases. The first phase creates a set of unique segments (some segments are duplicated), and each addition to the set of unique elements is enclosed by a transaction. In the second phase, each thread tries to remove a segment from a global pool of unmatched segments and add it to its partition of currently matched segments. The access to the global pool are also enclosed by a transaction.

– **Kmeans**: This benchmark groups objects in a N-dimensional space into K clusters, and it is usually used to partition data items into related subsets. In each iteration, the update of the cluster center is protected by a transaction. The amount of contention depends on the input parameters, the read and write sets are relatively small, and the total time spent on transactions is low.

– **Ssca**: Scalable Synthetic Compact Applications 2 (SSCA2) [11] is comprised of four kernels, but the STAMP implementation focuses only on one of them. This kernel constructs a graph data structure using adjacency arrays and auxiliary arrays. Transactions are used to protect the access to adjacency arrays, and since this action is relatively small, not much time is spent on transactions. Additionally, the length of the transactions and the sizes of their read and write sets is also small, as well as the amount of contention.

– **Yada**: This benchmark implements an algorithm for Delaunay mesh refinement [144]. The main structures are a graph to store all the mesh triangles, a set with the boundary segments and a task queue with the elements that need to be refined. In each step of the algorithm, a triangle is removed from the queue, its retriangulation is performed on the mesh and the new triangles formed are added to the work queue. The transactions are used to enclose the access to the queue, and as almost all the execution time is spent recalculating the retriangulation, this benchmark has relatively long transactions and almost all of the execution time is spent on them. This benchmark also has large read and write sets and moderate contention.

– **Bayes**: This benchmark implements an algorithm for learning the structure of Bayesian networks. The bayesian network is represented as a directed acyclic graph with a node for each variable and an edge for each conditional dependence between variables. The algorithm implemented gradually learns dependencies among variables by analyzing the observed data. A transaction is used to protect the calculation and addition of a new dependency. "Bayes" spends almost all its time in transactions, which have large read/write sets and high contention.

2.2.3 PARSEC

The Princeton Application Repository for Shared-Memory Computers (PARSEC) [16] is a benchmark suit to study multicore processors, which unlike SPLASH-2 or STAMP, includes applications in recognition, mining and synthesis (RMS). Some of the benchmarks are from Intel ("Blackscholes", "Bodytrack", "Facesim", "Fluidanimate", "Raytrace" and "Swaptions"), some of them from the Princeton University ("Ferret", "Canneal", "Dedup" and "Streamcluster"), and others are based on well known applications ("VIPS" and "x264") .

PARSEC benchmarks are written in C language, and the critical sections are enclosed with locks for the simulation of Pacman.

Below we describe all the PARSEC benchmarks used in this thesis.

– **Blackscholes**: This application calculates an estimation of the current value of European options with the Black-Scholes formula, the derivative of a partial differential equation (PDE) [17] which governs the price of the option over time.

– **Bodytrack**: This is a computer vision application which tracks a 3D pose of a human body with multiple cameras through an image sequence [13] [43].

– **Facesim**:This application (originally developed by the Stanford University) takes a model of a human face and a time sequence of muscle activations and computes a visually realistic animation of the modelled face by simulating the underlying physics [156] [165]. The goal is to create a visually realistic result.

- **Ferret**: This application is based on the Ferret toolkit which is used for content-based similarity search of feature-rich data such as audio recordings, digital images, sensor data, 3D shapes and so on [97].
- **Fluidanimate**: This application uses an extension of the Smoothed Particle Hydrodynamics (SPH) method to simulate fluid dynamics for interactive animation purposes [111].
- **Raytrace**: This application renders an animated 3D scene for real-time animations (such as computer games). Ray tracing is a technique that generates a visually realistic image by tracing the path of light through a scene [177]. This application is also included in SPLASH-2 benchmarks (Section 2.2.1).
- **Swaptions**: The swaptions application uses the Heath-Jarrow-Morton (HJM) framework to price a portfolio of swaptions. The HJM framework describes how interest rates evolve for risk management and asset liability management [69] for a class of models. It employs MonteCarlo simulation to compute the prices.
- **Vips**: This application is based on the VASARI Image Processing System (VIPS) [103]. The benchmark is an imaging processing system, which includes fundamental image operations such as an affine transformation and a convolution.
- **x264**:This application is an H.264/AVC (Advanced Video Coding) video encoder. It is based on the ITU-T H.264 standard which is now also part of ISO/IEC MPEG-4. It improves on previous video encoding standards with new features such as increased sample bit depth precision, higher-resolution colour information, variable block-size motion compensation (VBSMC) or context-adaptive binary arithmetic coding (CABAC).
- **Canneal**: This kernel uses cache-aware simulated annealing (SA) to minimize the routing cost of a chip design [14]. SA is a common method to approximate the global optimum in a large search space.
- **Dedup**: This kernel compresses a data stream with a combination of global compression and local compression in order to achieve high compression ratios. Such a compression is called ’deduplication’.

- **Streamcluster**: This kernel solves the online clustering problem [85]: for a stream of input points, it finds a predetermined number of medians so that each point is assigned to its nearest center. The quality of the clustering is measured by the sum of squared distances (SSQ) metric. It is used in network intrusion detection, pattern recognition and data mining.

2.2.4 EigenBench

"EigenBench" [73] is a lightweight, flexible and powerful synthetic benchmark designed to evaluate and understand TM systems by forcing different TM orthogonal characteristics. These characteristics are the basics to understand TM behavior, and it is also useful to reproduce some TM pathologies, or evaluate corner cases that are not easily reachable with standard applications. These characteristics are the following:

- **Concurrency**: Number of concurrently running threads.
- **Working-set size**: size of the frequently used memory.
- **Transaction length**: Number of shared accesses per transaction.
- **Pollution**: Fraction of shared writes to shared accesses.
- **Temporal locality**: Probability of repeated address per shared access.
- **Contention**: Probability of conflict of a transaction.
- **Predominance**: Fraction of shared access cycles to total execution cycles.
- **Density**: Fraction of non-shared cycles executed outside transactions to total non-shared cycles.

There exists an actual mapping from a real application to a set of these orthogonal characteristics [73]. Furthermore, "EigenBench" is written in C, and it uses the same TM API as used by STAMP. In our case, we use a certain set of C Macros that maps TM accesses with RSTM. We use this benchmark to explore some TM scenarios in our evaluation of asymmetric *FlexSig* (in Chapter 6).

2.2.5 Other Benchmarks

We use two additional benchmarks to evaluate Pacman, which we describe below.

Apache

"Apache" [50] is the most used web server software. At a high level, the Apache server architecture is composed of a core that implements the most basic functionality of a web server and a set of standard modules that actually service the phases of handling an HTTP request.

Sphinx3

"Sphinx3" is a decoder for speech recognition research written in C [150]. It includes both an acoustic trainer and various decoders, i.e., text recognition, phoneme recognition, N-best list generation, etc.

2.3 Conclusions

We present in this chapter the simulator environment and the benchmarks used in this thesis. The techniques of simulation used are very well know, and very popular for analyzing computer architecture innovations. Furthermore, the benchmarks represent a high variety in their characteristics, and also are very representative of the current workloads which can be potentially used in real world environments.

CHAPTER 3

REDUCING THE USE OF SIGNATURES IN A HTM SYSTEM

Hardware Transactional Memory (HTM) systems have been very popular due to their performance advantages. However, this performance improvement comes at the cost of increasing the hardware resources. Typical hardware additions in a HTM are cache add-ons, cache coherence support, special caches for speculative data or hardware signatures.

In this chapter we propose a method to save resources in a HTM system. Specifically, we propose a simple Cache Filtering Mechanism (CFM-TM) for HTM systems [123], which acts like a filter by managing part of the write set of the application, with the aim of reducing the use of signatures (see Section 1.5) and log information in the transactional memory baseline system. In addition, to fully take advantage of this method, the CMP system should have signatures with different sizes, or, preferably, it should have signatures of variable size using a system like the schemes proposed in Chapters 5 and 6.

We test our CFM-TM with LogTM-SE [183] as the baseline system (Section 1.4.2), because it is a popular implementation that uses signatures for conflict detection. Based on our experimental evaluation, by using this filtering mechanism the size of signatures are significantly reduced with no significant degradation of performance (in one of the benchmarks used in the evaluation, there is even an important improvement).

3.1 System Architecture

The system architecture is composed by the LogTM-SE implementation [183] as the baseline system, and our CFM-TM attached to this system.

The general vision of the architecture is a multicore system with private L1 data caches and shared L2 cache memory. The caches are inclusive, and therefore, if a line is stored in the L1 cache, it has to be also stored in the L2 cache. In this environment the LogTM-SE HTM system implements an eager conflict detector that detects conflicts among transactions with signatures, and an eager version manager, which logs old versions of transactional writes in per-thread private memory. Furthermore, LogTM-SE uses the same structure and tags for the cache lines, but introduce some changes in a conventional cache coherence protocol (it is expanded with sticky states [183]). More details about LogTM-SE can be found in Section 1.4.2.

The goal of CFM-TM is to manage transactional writes faster, and free the LogTM-SE system from these operations. The architecture of CFM-TM is based on modifications of the private L1 cache memory, cache coherence protocol and the replacement algorithm. The baseline cache coherence scheme used in this work is the directory-based MESI protocol explained in Sections 1.1.3 and 1.4.2, with the directory placed at the shared L2 cache.

3.1.1 Managing Transactional Writes with CFM-TM

The CFM-TM filters some transactional writes to the LogTM-SE base system. This filter is implemented with several changes in the base architecture, which we describe below.

The *WTx* Bit

CFM-TM adds a *WTx* bit at every L1 private cache line with to aims: detecting conflicts and hosting speculative data in L1. A transactional modified line is managed by the CFM-TM if its *WTx* bit is active (set to one). This bit is only accessible by the local core, and when it is set to 1, the cache line can not be evicted from L1 during the transaction (to maintain both versions of the data, the speculative version in L1, and the old version in L2). When a core is trying to access to a remote transactional line managed by the CFM-TM (*WTx=1*), the filter detects a conflict of data that is managed by the contention manager.

If a line has the *WTx* unset, it is managed by the transactional baseline system with the regular protocol (the line is not managed by the CFM-TM filter). However, a cache line with the *WTx* bit set to 1 has to be in state Exclusive or Modified (the copy of the line has to be present only in this L1 cache).

Associativity Reduction

CFM-TM works at the cost of reducing some cache capacity for other cache lines not hosted in the filter. When the *WTx* bit is set in a cache line, this line can not be evicted from the L1 private cache until the transaction commits or aborts, and because of this, the maximum number of speculative data in an associativity set has to be limited. This limit, which sets up the maximum amount of transactional data in the associativity set that can not be evicted, is called *MNW* (Maximum Non-evicted Ways). The *MNW* value is bounded by the number of ways of the associativity set of the L1 cache. According to our tests, a value of *MNW* of 25% of the associativity ways is the most appropriate. The value of *MNW* can be modified by software through a new instruction. Since the transactional data with the *WTx* bit set is not evicted, the number of ways of the associativity set might be dynamically and temporary reduced to other data lines. The *MNW* parameter can be changed dynamically at execution time (before each transaction starts). If *MNW* is zero, the filter never hosts transactional writes, and it remains deactivated.

Check for Conflicts

Figure 3.1 shows a flowchart that illustrates the checking for conflicts in the write set using CFM-TM and LogTM-SE. First, the value *MNW* is checked to know if the CFM-TM is active. In case the value of *MNW* is zero, the filter is deactivated, and the transactional memory system use the baseline LogTM-SE system. In case the value *MNW* is not zero, the filter is active, and it has to check if the *WTx* bit associated with the cache line is set to one. In affirmative case, a conflict is produced, and it is not necessary to activate the conflict detection system of the baseline TM system. In case the *WTx* is zero, the line is not managed by the CFM-TM, and the LogTM-SE system has to check the signatures looking for data conflicts.

Managing L2 Transactional Data

Since the memory cache system is inclusive, if a transactional write can not be evicted from L1 (*WTx* bit is set), then it can not be evicted from L2. Therefore, an associativity set of the L2 cache memory might be filled with non-evictable cache lines if we do not prevent

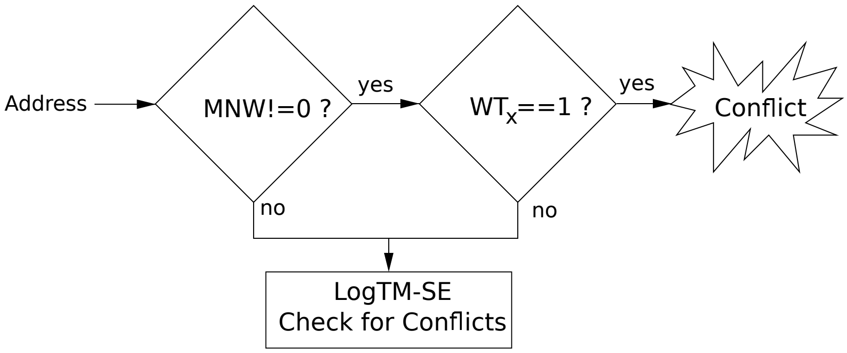


Figure 3.1: Check for conflicts in the write set in a LogTM-SE system using the CFM-TM filter.

this situation. To solve this, a control bit *WTxL2* analogous to *WTx* is defined in the directory, and a new parameter *MNWL2* (analogous to *MNW*) is managed by the system. The parameter *MNWL2* (Maximum Non-evicted Ways in L2) indicates the maximum amount of transactional data in the associativity set that can not be evicted in L2. If the *WTxL2* bit is set in the L2 cache, it means that the line is transactional, and the corresponding speculative data is in L1. The *WTxL2* parameter is also used for preventing a speculative line from being evicted when the eviction starts at the L2 cache controller.

Transactional Write Actions

Figure 3.2 shows a flowchart that illustrates the actions taken when a transactional write is produced. As we see in the figure, first, the system checks if the CFM-TM filter is active (*MNW* greater than zero). If the filter is deactivated, the system uses directly the LogTM-SE system, but when it is active, in order to check if CFM-TM can host the transactional cache line, the L1 cache controller checks if the limit of *MNW* speculative writes has been reached. In case the limit is reached, the LogTM-SE system takes the control. Otherwise, the L1 cache controller requests the L2 for the line (in case the line is in invalid state) or informs the directory that needs the exclusivity of the line (in case the line is in shared state). When the directory responds to the requester, it informs if the L2 can host the speculative line according to the *MNWL2* parameter. When a transactional write is produced in a line that is in an Exclusive or Modified state with its *WTx* bit unset, the baseline coherence protocol does not make any request because the state does not change (with respect to the directory).

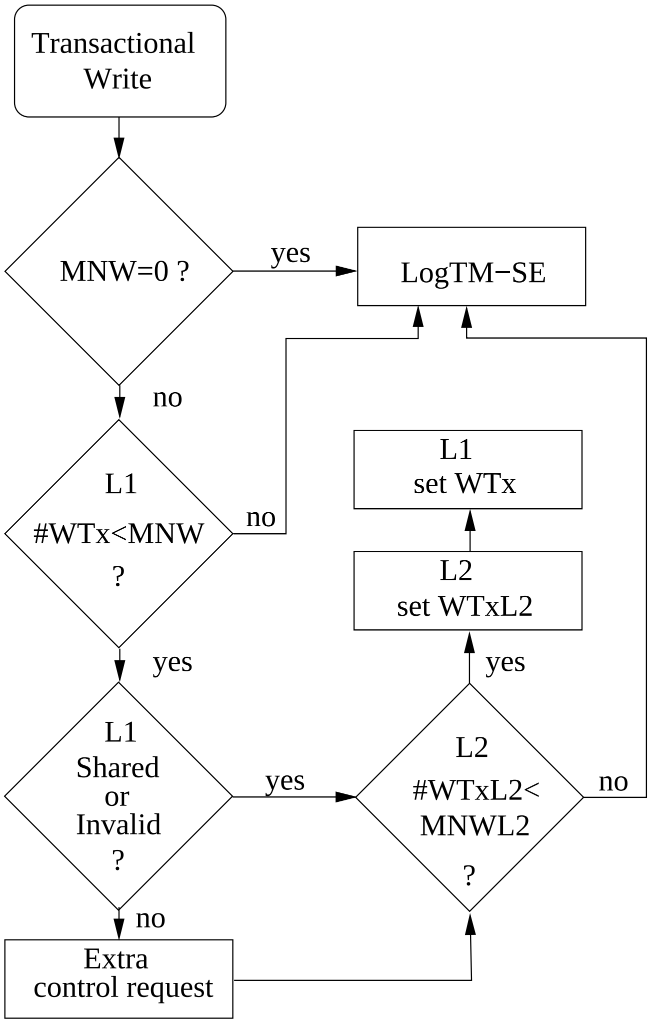


Figure 3.2: A transactional write action in the LogTM-SE system using the CFM-TM filter.

Furthermore, in order to test the conditions established by the *MNWL2* parameter, an extra control request to the directory is necessary.

3.1.2 Replacement Algorithm

To keep speculative data within the private L1 cache, the LRU (Least Recently Used) replacement algorithm is modified so that the cache lines having the *WTx* bit set are not evicted. The LRU algorithm changes slightly: when the algorithm finds that the line to be evicted (the least recently used) has the *WTx* bit set, the line is marked as the most recently used, and it searches the next least recently used line. If the line does not have the *WTx* bit set, it can be evicted.

This scheme is analogous for replacements at the L2 cache. When a core request a line that it is not in the cache, and has to make room in L2 because there is not free space, the replacement algorithm can not evict a transactional line that holds the old data in L2. The replacement algorithm is the same than in the L1 cache and it keeps the data that had been written during the transaction in the caches till the end of the transaction (when the *WTx* bits are all set to zero).

3.1.3 Commit and Abort Actions

When a transaction commits, the *WTx* bits in the corresponding private L1 cache are flash cleared by the CFM-TM system, and consequently all the speculative data transits to non-speculative and is accessible by all cores. Figure 3.3 shows an example how commits are managed by the CFM-TM system by just changing the *WTx* bits to zero.

In an abort all lines with the *WTx* bit set are marked as invalid, the M bit (corresponding to M and E states) is cleared, and then the *WTx* bits are flash cleared. All the speculative data is discarded. Figure 3.3 shows all these commit actions. In this situation the directory is not warned (silent abort), and some actions are added to manage this situation: when a request forwarded by the directory arrives at a core that has that line in an invalid state, it means that a silent abort was produced in that core recently. In this case, the core that receives the request, responds to the directory informing that it is no longer the owner of the line. Then, the directory updates the state information and send to the original requester the valid line hosted at the L2 shared cache.

Usually, not all the transactional data can be managed by the filter, and therefore, the LogTM-SE system has to manage the data that overflows the filter. Hence, the commit and abort actions has two phases: the CFM-TM phase (very fast, 1 cycle), and the LogTM-SE phase, that takes more time, because it is required to clean signatures and in case of aborts, restore old data hosted in the log.

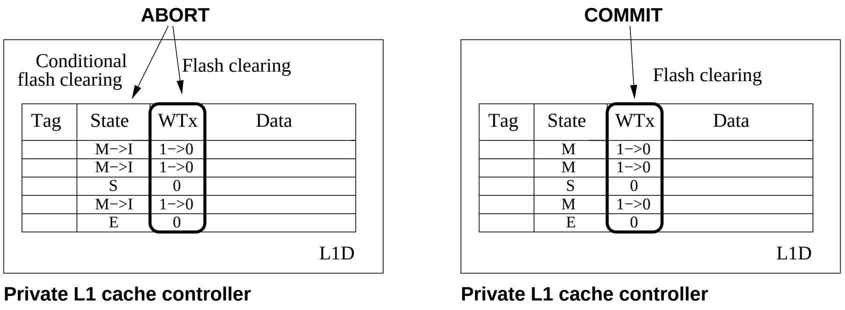


Figure 3.3: Abort and commit actions.

3.1.4 Limitations

The main drawback of the CFM-TM scheme is that it is not virtualizable, because software access to transactional information is not allowed. In case of context switching, paging or any virtual action is produced, the running transactions that used this filtering must be aborted.

Also, because CFM-TM fix the speculative data to the caches, and these lines can not be evicted, a wrong dimensioning of the *MNW* parameter (by the programmer) can lead to a slowdown of the system, usually because of the limited associativity for other cache lines. However, deadlock can not be produced, as there is a mechanism that prevents to set the *MNW* value with values equal or greater than the number of ways of and associativity set.

3.1.5 Interactions between CFM-TM and LogTM-SE

To incorporate CFM-TM to LogTM-SE it is necessary to make the changes described in Sections 3.1.1 and 3.1.2 to the cache array, the coherence protocol and the replacement algorithm. The implementation of CFM-TM for other decoupled TM systems is very similar for systems with an eager conflict detection policy.

CFM-TM affects LogTM-SE when a transactional write is produced (see Figure 3.2). Before inserting an address into the write signature and logging the old version of data, the filter checks if the transactional write can be hosted in the L1 private cache without eviction, and if this is the case, LogTM-SE does not manage that write. Moreover, before checking read/write signatures to detect if a conflict is produced, it is necessary to know if the line has

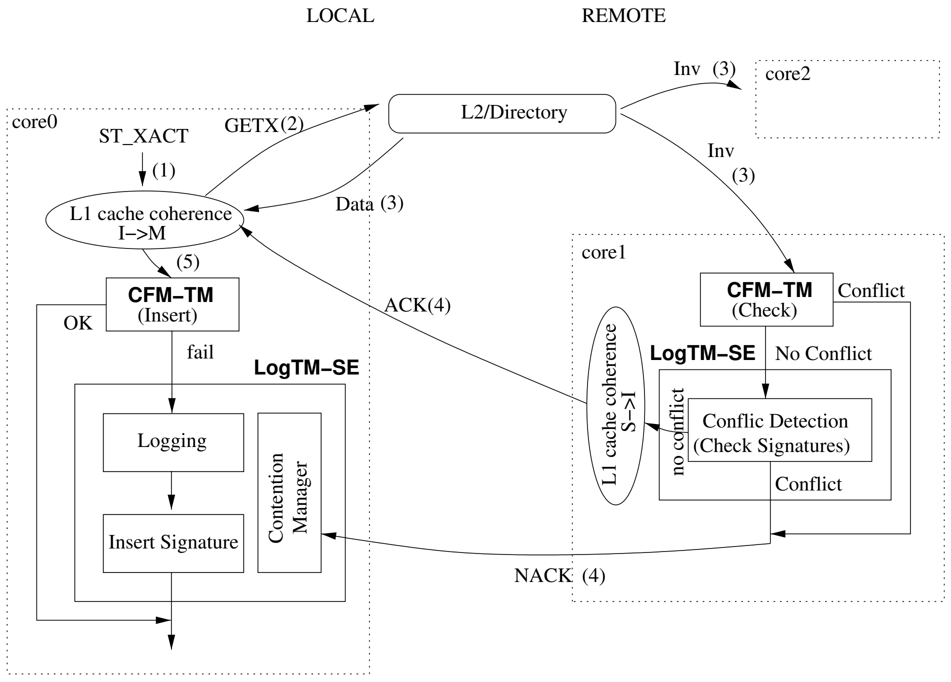


Figure 3.4: Interaction between CFM-TM and LogTM-SE in a transactional store miss. The missed line is shared by other two caches.

its *WTx* bit set, and in this case, because the line is a transactional write, a conflict is produced without checking signatures (see Figure 3.1).

Next, we explain some possible scenarios where a transactional store miss or a transactional store hit is produced.

Store Miss

Figure 3.4 illustrates the actions in a transactional store miss in core 0, when it is shared by two cores (core1 and core2):

- In (1), a transactional write (ST_XACT) is produced in core0 over a invalid line.
- The core0 sends a request (GETX) to the directory asking for the line in exclusivity (2).

– The directory checks that this line is shared by core1 and core2, and it sends an Invalidation signal (Inv) to these cores, and sends the data to core0 (3).

– core1 and core2 check for conflicts (in the CFM-TM and (if necessary) in the both LogTM-SE signatures), and in case of no conflict these cores, invalidate the local copies of the line, and send and ACK to core0 (4) . When core0 receives the ACK, the line make the transition to Modified (M) state.

– In case a conflict is detected in core1 or core2, a NACK is send to core0 (4), that initiates the conflict resolution in the contention manager.

– Finally, if no conflict is detected, the core0 inserts the speculative data in the filter in case there is enough space, or in the LogTM-SE otherwise (5).

The actions to be taken in the same situation, but with the local line in the Shared state, are very similar.

The case of a transactional store miss in core0, when the line was modified by core1, is illustrated in Figure 3.5:

– In (1), a transactional write (ST_XACT) is produced in core0 over a invalid line.

– Core0 sends a request (GETX) to the directory asking for the line in exclusivity (2).

– The directory checks that such line might be dirty and owned in exclusivity by core1, and it sends and Invalidation (Inv) to that core (3).

– core1 checks for conflicts. In case of conflict, core1 sends a NACK to core0, that activates its contention manager (4).

– In case of no conflict, the core0 receives the data from the core1 (4).

– Finally, the speculative data is inserted in the filter if there is enough space, or it is handled by the LogTM-SE otherwise (6).

Store Hit

Figure 3.6 illustrates the actions in a transactional store hit when the line is already in M state in the local cache (core0):

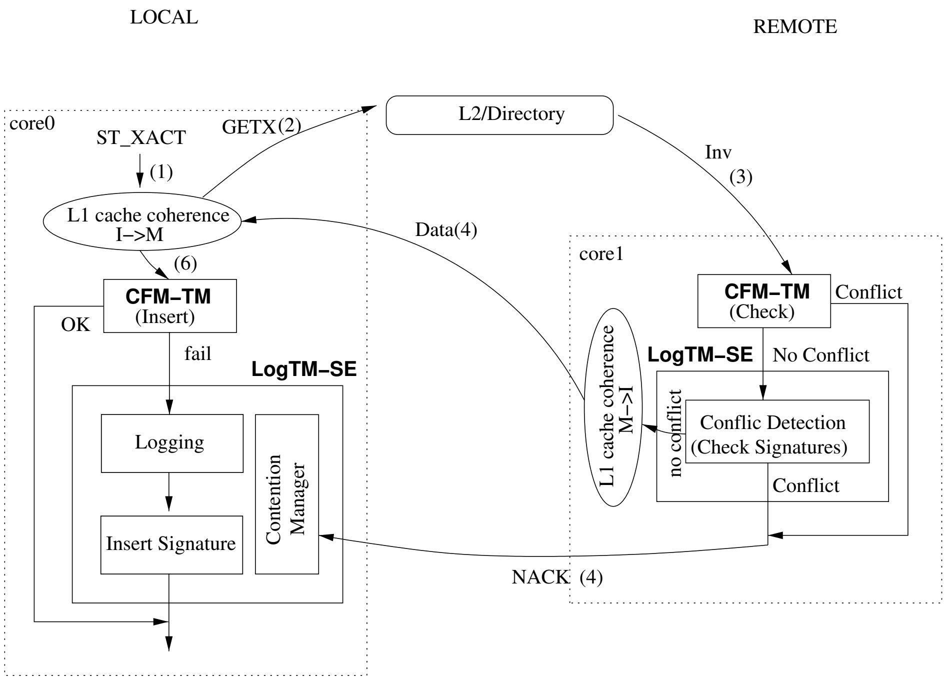


Figure 3.5: Interaction between CFM-TM and LogTM-SE in a transactional store miss. The missed line is in other cache in M state.

- In (1), a transactional write (ST_XACT) is produced in core0 over a modified line.
- The CFM-TM has to send an extra request to L2 (2), to inform that the line is transactional now (and set the *WTxL2* bit)
- If an ACK is received from L2 (3), the line is managed by the CFM-TM if possible (enough room in L1 cache). If a NACK is received from the L2 cache (3), then LogTM-SE handles directly the line.

Commits and Aborts

In a commit, LogTM-SE has to reset the log pointer, and clean the read and write signatures, and in an abort it has to restore old versions hosted in the log and clear the read/write

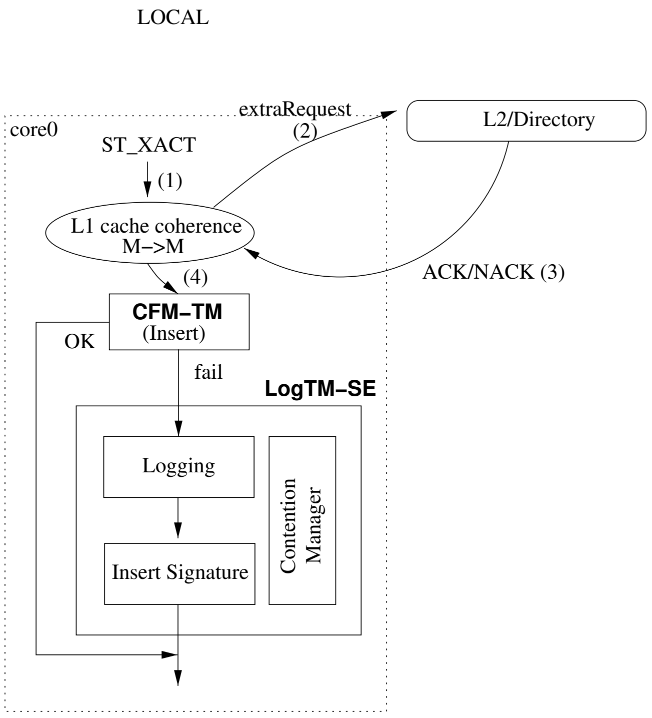


Figure 3.6: Interaction between CFM-TM and LogTM-SE in a transactional store hit. The line is already in M state in the local cache.

signatures. Moreover, LogTM-SE has to give the order to commit or abort to the CFM-TM system if it is activated.

LogTM-SE performs commits fast, but aborts are slow. On the other hand, CFM-TM performs fast commits and fast aborts. When the CFM-TM system is incorporated to LogTM-SE, commits are fast anyway, but aborts might be faster, due to the transactional writes managed by CFM-TM.

The temporary reduction of associativity in certain cache sets may increase the cache misses of data, but as we show in the evaluation section, this is compensated with the misses saved since logging in per-thread private memory is not required. For very large transactions, with a huge read and write set, the system may perform better if the CFM-TM filter is deactivated. However, the filter works well even with workloads that are considered large (see the evaluation in Section 3.3).

3.2 Signatures

As mentioned above, LogTM-SE uses read and write hardware signatures (see Section 1.5) to detect conflicts. When LogTM-SE works with CFM-TM, the write signature may be reduced, because CFM-TM manages some transactional writes which does not reach the write signature of LogTM-SE. This contributes to use a smaller write signature without increasing the false positives rate. Other papers deal also with this problem, like Notary [184], where new techniques are proposed for reducing hardware cost and false conflicts (by privatization) that result in more efficient signatures.

Signatures have the problem of having a fixed size and being not scalable. Specifically, a single size is not well suited to all kind of applications, i.e. signatures of 1024 bits might be oversized for a benchmark with small transactions [178], but too small for long size transactions [106]. To solve this problem, more signatures with different sizes could be provided, so that the application chooses the signature of minimum size that allows good performance.

One contribution of this thesis proposes a new module of signatures, called *FlexSig* (see Chapter 5 and Chapter 6). This work goes further, and try to adapt the resources available for signatures to the demand of the request. To achieve this, priorities can be established depending on the needs of the requesters.

FlexSig is an appropriate solution to manage the read and write signatures of a TM system with a CFM-TM filter, because it provides a flexible module that can assign few resources to the write signature, and to use the remaining resources for other purposes (for instance, a bigger read signature and the read and write signatures of other transactions). *FlexSig* can change the assigned resources in run time, it allows a more efficient use of the signature resources and it reduces the false positive rate (which improves the overall performance), which make it a good option for combining with the CFM-TM filter.

3.3 Evaluation

In this section we evaluate the proposed CFM-TM filter. To evaluate the system, we compare LogTM-SE with and without CFM-TM.

3.3.1 System Model

We evaluate CFM-TM and LogTM-SE by using GEMS [102] and the Simics [98] full system simulator (the framework described in Section 2.1.1) with the configuration of Table 2.1.

The sizes of signatures are between 128 bits and 8192 bits, and are chosen depending on the application, trying to obtain a reduced false positive rate, with the minimum signature size. The size of signatures are chosen in order to maintain a rate of false positives similar in both systems.

3.3.2 Workloads

We use three STAMP benchmarks [106] [1] ("Vacation", "Intruder" and "Labyrinth"), described at Section 2.2.2, and the "Barnes" SPLASH-2 workload [178], described in Section 2.2.1. Table 3.1 shows the workload characteristics, indicating the number of transactions and the average sizes of the read and write sets. Table 3.2 shows the inputs for each benchmark evaluated.

Table 3.1: Workload characteristics.

Benchmark	#Tx	Av.RS	Av.WS
Intruder	11224	7.23	3.36
Barnes	2330	5.8	4.4
Vacation	24776	19.7	3.6
Labyrinth	158	136.8	90.82

Table 3.2: Benchmark inputs.

Benchmark	INPUT
Intruder	-a10 -l4 -n2038 -s1
Barnes	4096 bodies
Vacation	-n4 -q60 -n90 -r16384 -t4096
Labyrinth	-i random-x32-y32-z3-n64.txt

[1] with Luke Yen's patches from the University of Wisconsin.

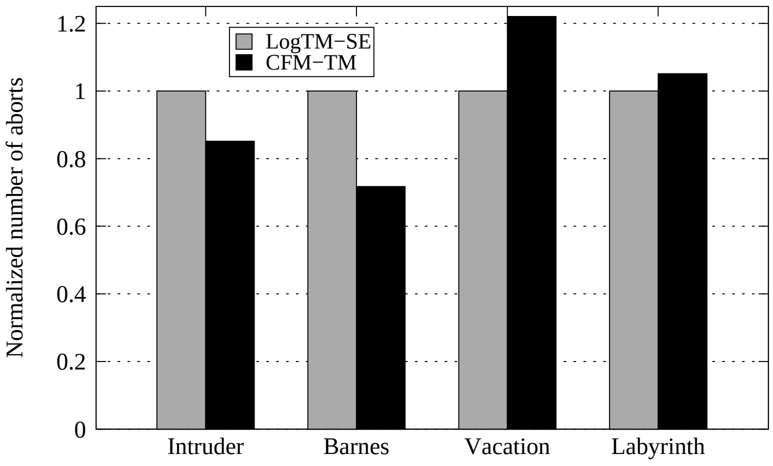


Figure 3.7: Normalized number of aborts.

3.3.3 Results

For each benchmark and system configuration we determined the minimum signature size that allowed a rate of false positives less than 1%. We obtain a general reduction in the size of the write signature, and for some benchmarks even a performance improvement. It is very difficult to calculate so precisely the size of the signatures for achieve a number of aborts equal in all the benchmarks. This variability in the number of aborts is reflected in Figure 3.7. Following this rule, as Table 3.3 shows, the write signatures are reduced 75% in 3 of the benchmarks (from 512 to 128 bits), and it is reduced a 50% for the "Labyrinth" case (from 8192 to 4096 bits).

Table 3.3: Read/Write signature's size configuration in LogTM-SE with and without CFM-TM.

Benchmark	LogTM-SE RS/WS	CFM-TM RS/WS	WS reduction
Intruder	512/512	512/128	75%
Barnes	512/512	512/128	75%
Vacation	1024/512	1024/128	75%
Labyrinth	8192/8192	8192/4096	50%

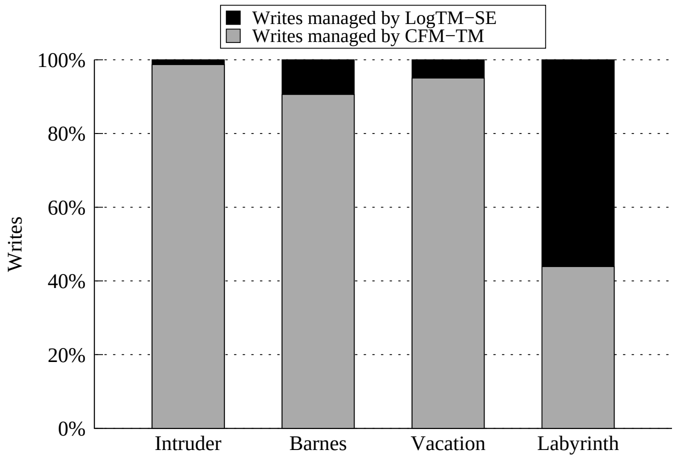


Figure 3.8: Write management distribution.

To illustrate the influence of CFM-TM in the system, Figure 3.8 shows the proportion of writes managed by CFM-TM, and the writes managed by the LogTM-SE system (when the CFM-TM can not manage those writes). As the figure shows, most of the writes are managed by the CFM-TM system in "Intruder", "Barnes" and "Vacation", and near 50% is managed in "Labyrinth".

Figure 3.9 shows the speedup of LogTM-SE + CFM-TM with respect to using LogTM-SE alone. Three of the benchmarks have roughly the same performance, whereas "Intruder" is more than 40% faster with the CFM-TM.

For the benchmark "Barnes" we obtain a reduction of the signature size by a factor of four with roughly the same performance. It has an average read/write set that CFM-TM manage without problems.

"Intruder" behaves specially well with CFM-TM. It is a high contention benchmark with an average read/write set that LogTM-SE doesn't manage well because it does not have a sophisticated contention management policy, leading to a high number of transactions that abort. When CFM-TM is active, performance is improved in more than 40% because the average write set of the benchmark is managed almost completely by the filter, which allows fast aborts. Furthermore, the write signature reduction is 75%.

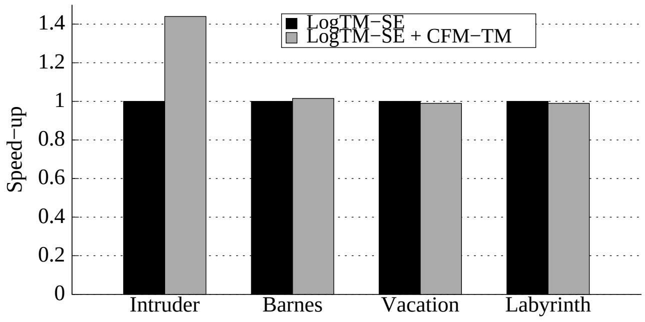


Figure 3.9: Speed-up of LogTM-SE + CFM-TM.

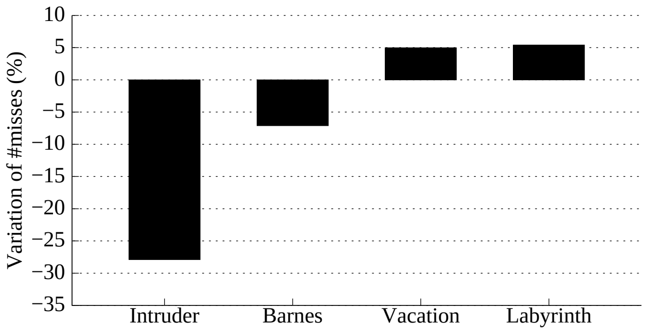


Figure 3.10: Variation of L1 cache misses when the CFM-TM is activated (with respect to LogTM-SE alone).

For benchmarks "Vacation" and "Labyrinth", with larger transactions, CFM-TM performance is slightly worse, but the signature size is reduced 75% for "Vacation" and is reduced 50% for "Labyrinth".

Figure 3.10 shows the variability in the number of L1 cache misses due to a reduced associativity by transactional execution. For "Vacation" and "Intruder" there is a small increase of 5% in the number of misses. However, for "Barnes" and "Intruder" even the number of L1 cache misses are reduced because the logging is not performed.

Cycle Breakdown

The cycle breakdowns provides detailed information about the execution time. Figure 3.11 shows the normalized cycle breakdown for each benchmark, with both configurations: the LogTM-SE alone and with the CFM-TM. The cycles are grouped in different phases: NON_TRANS represents the cycles used in non transactional code, BAD_TRANS represents the transactional cycles that were wasted in transactions that finally aborted, GOOD_TRANS are the good transactional cycles that lead to successful transactions, ABORTING are the cycles expended in the abort process, COMMITTING are the cycles expended in commits, BACKOFF are the backoff cycles (randomized number of cycles to reduce contention after aborts), BARRIER are the cycles spent in barriers and STALL are the cycles that transactions are stalled (when a conflict is produced, and trying to resolve the conflict without aborting).

Figure 3.12 shows the normalized cycle breakdown of each benchmark, showing the specific phase in the *x* axis. The values are normalized to compare the time spent in each situation with and without the CFM-TM. In these normalized graphs we can appreciate much better the improvement or deterioration of the behavior in each specific phase for each benchmark.

Figure 3.12 (a) shows the results for the "Intruder" benchmark. The cycles are reduced in BAD_TRANS, ABORTING, BACKOFF, BARRIER and STALL phases because the number of aborts is less (see Figure 3.7) as well as the time spent in them. In the "Intruder" benchmark there is a barrier at the end of the transactional processing, which explains that the BARRIER time is also decreased (if there are less aborts, the barrier has to wait less time for the transactions).

Figure 3.12 (b) shows the results of the "Barnes" benchmark. We see that the BAD_TRANS are increased with CFM-TM, because without the filter, some false positives are detected before the actual conflict is produced, and therefore these transactions abort before, and save some cycles. The GOOD_TRANS variability may be caused because fluctuations in the benchmark and the small relative time spent in transactions (see Figure 3.11). The ABORTING and BACKOFF reduction time are caused because the action of the CFM-TM filter.

Figure 3.12 (c) shows the results for the "Vacation" benchmark. ABORTING cycles are less with CFM-TM (as expected), and it has worse behavior in BAD_TRANS, BACKOFF and STALL, because the number of aborts is slightly superior.

Figure 3.12 (d) shows the results for the "Labyrinth" benchmark. We see again that the biggest difference is with the ABORTING cycles, due to the CFM-TM.

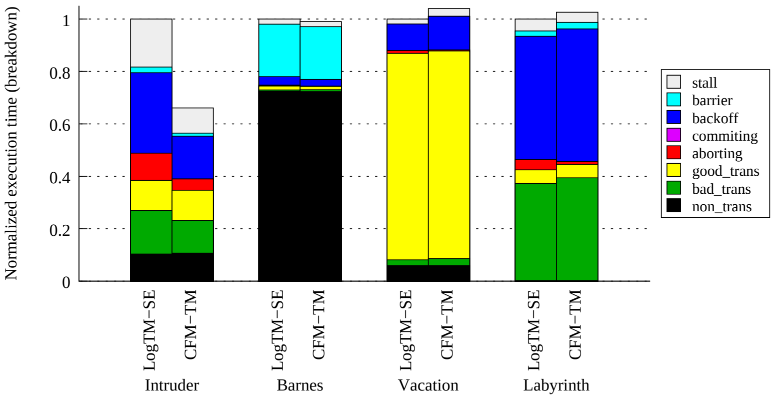
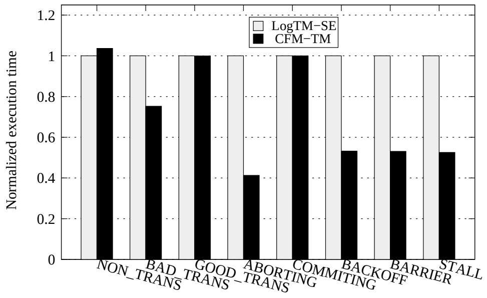
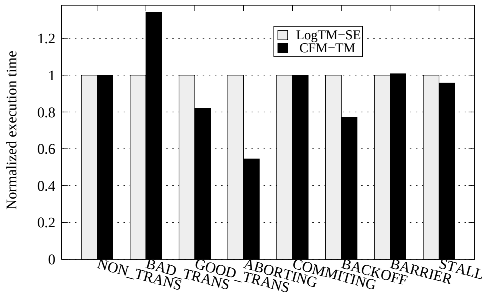


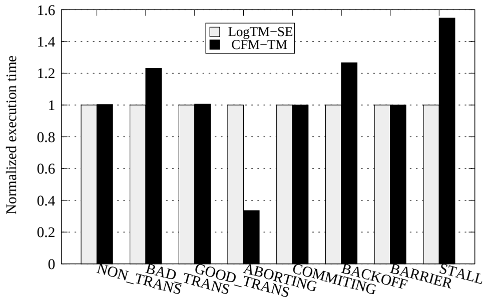
Figure 3.11: Normalized breakdown of execution cycles.



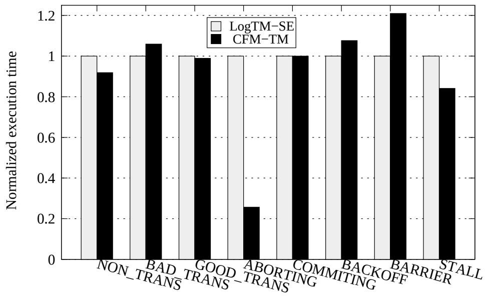
(a) "Intruder" breakdown.



(b) "Barnes" breakdown.



(c) "Vacation" breakdown.



(d) "Labyrinth" breakdown.

Figure 3.12: Benchmarks breakdown.

It is clearly visible from Figure 3.12 that the number of cycles spent in the abort process is drastically reduced.

3.4 Related Work

The CFM-TM scheme tries to reduce write signature utilization and enhance the performance of TM systems that are decoupled from caches. In our paper we choose LogTM-SE as baseline system, because it is a very representative decoupled HTM system and it is easily virtualizable. Moreover, some HTM systems use the cache buffering capability in a similar way as CFM-TM (FlexTM [155] and UTM [8]).

FlexTM is a flexible HTM that, among other things, uses signatures for conflict detection, uses L1 as a buffer for speculative data, and uses a per thread log when the cache overflows. The difference with CFM-TM is that FlexTM uses signatures all the time to detect conflicts, meanwhile CFM-TM detects conflicts using a *WTx* bit in L1 private cache.

Unbounded Transactional Memory (UTM) uses a special structure held in memory to log old versions, and add two bits for each line that may be visible in all the memory hierarchy, in order to track transactional data. If the transaction fits in cache, the system does not log old versions (similar to CFM-TM) unless the line is evicted.

CFM-TM differs from the previous HTM systems by hosting transactional data in L1 private cache until the transaction commits or aborts, that allows the system to use signatures in a conservative way (they are used as little as possible).

FASTM: A Log-based Hardware Transactional Memory with Fast Abort Recovery

FASTM [95] proposed a similar solution to CFM-TM, and it was published at the same time. It also proposes to use the cache memory to maintain the speculative data whereas the old data remains in higher levels of the cache memory. The difference with CFM-TM is that FASTM does not fix the transactional data to the cache, and it does not reduce the associativity temporally. Instead, when it has to evict a transactional write, the address is inserted in the signature. FASTM achieves a speed-up of 43% compared to LogTM-SE using the same signature sizes.

3.5 Conclusion

In this paper we presented CFM-TM, a Cache Filtering Mechanism for Transactional Memory systems in order to reduce the hardware resources of the baseline TM system without reducing performance.

We evaluate CFM-TM with LogTM-SE as baseline system using some benchmarks. We showed that in most of the cases the system performs better when the CFM-TM is activated, and what is more important, it is possible to reduce the size of the write signature used by LogTM-SE. CFM-TM can be deactivated by software at any time if virtualization is needed. To fully take advantage of the proposed filter, the multicore processor system should support a flexible management of signatures, and allowing signatures of variable size, such as our proposals in Chapters 5 and 6.

CHAPTER 4

TOLERATING ASYMMETRIC DATA RACES WITH A HARDWARE SIGNATURE MODULE[1]

Parallel debugging is one of the keys to improve the productivity of parallel programmers. Concurrent errors are difficult to detect and debug, and it is an important source of low productivity. To solve this problem, many approaches have been proposed to deal with different kind of concurrent bugs. Data races are one of the most frequent causes of bugs, and they have received a significant attention by the research community [113] [132] [94] [151] [48] [49] [92] [172].

However, an important type of data races that have not received much attention are *Asymmetric* data races, described in Section 1.3.2. In these type of races, the state of well-tested, correct threads is corrupted by racing threads from external, typically third-party code.

Figure 1.11 shows an example of an asymmetric data race, where a correctly synchronized thread (the *safe thread*) is corrupted by a thread that is not (the *unsafe thread*). In this example, the safe thread (T1) access to the content of a pointer in a correctly synchronized critical section, whereas T2 access the same pointer without synchronization, which produces a unpredictable behavior in T1 when it tries to update the contents of the pointer. The idea of this work is to *detect* when an *unsafe thread* tries to modify the state being accessed by a *safe thread* and *prevent* it from doing so. The result is a correct critical section. Correctness means

[1]This work was developed at the University of Illinois at Urbana-Champaign in collaboration with the members of the I-ACOMA group. Specifically, my contribution was: active participation in the discussion of the general and advanced ideas, evaluation and experimental results, as well as writing and reviewing the resulting conference paper.

that the value of shared variables inside the critical section should not be changed between the first access inside the critical section and the end of the critical section.

Current schemes to detect and tolerate asymmetric races are software based [140, 137]. They prevent the corruption of the state in a critical section by copying the data accessed in the critical section to a local area, or by setting address translation protection bits in the core. However, this requires substantial execution overhead.

In this chapter we propose the first scheme to detect and tolerate asymmetric data races in hardware. The approach, called Pacman [132], induces negligible execution overhead and requires minimal hardware modifications. In addition, compared to past software-based schemes, Pacman eliminates deadlock cases. Pacman is based on using hardware signatures (see Section 1.5) to detect the asymmetric races.

The rest of the chapter is organized as follow: Section 4.1 introduces the problem, Section 4.2 describes Pacman architecture, Section 4.3 is focused on the hardware implementation, Section 4.4 evaluates Pacman, Section 4.5 discusses related work and Section 4.6 presents some conclusions.

4.1 Asymmetric Data Races in Real World

The focus of this chapter is a common and likely harmful type of race condition called Asymmetric data race. This is a data race where at least one of the racing threads is inside a synchronization-protected critical section [137] [140]. In addition, we are interested in efficiently tolerating them in production runs.

Harmful asymmetric data races are common in the real world. To assess their frequency, we examined 50 harmful data race bugs from bug libraries of open source software and from Microsoft reports. We define harmful as being a bug that the user wants to be fixed (as opposed to the many data races explicitly created by the programmer for performance). Of the 50 harmful races, we found 10 that are asymmetric. This is a significant 20%. They are shown and described in Table 4.1.

The high frequency of asymmetric data races is confirmed by Microsoft researchers in [137] [140], who claim that they frequently encounter them in software development. They provide two intuitive sources of asymmetric data races. One source is the case of code developed by reliable software developers that has to share memory state with less-tested code developed outside of the house (e.g., various device drivers). A second source is legacy code.

Application	Source	Description	Outcome
Apache 1.1 Beta	Bug number 1507	AppenderAttachableImpl object should be protected by synchronization in AsyncAppender.getAllAppenders	Exception
MySQL6.0	Bug number 48930	lock state is updated by two different threads holding different mutexes	System hangs
Mozilla-JS	Bug number 622691	The write to cx→ runtime→ defaultCompartmentIsLocked is not consistently protected by the lock	Incorrect result
Mozilla-XPConnect	Bug number 557586	One thread sets gLock to null before another thread drops the lock	Segmentation fault
Mozilla-Video/Audio	Bug number 639721	mInfo is written by nsBuiltinDecoderReader without its lock while mInfo is read from HaveNextFrameData with a lock	Incorrect result
Pbzip2-0.9.4	Paper [179] [186]	main() frees fifo→mut without protection	Segmentation fault
Windows Kernel	Case study 2 in slides of [76]	Two threads access the same structure with different mutexes	Incorrect result
Windows Kernel	Case study 3 in slides of [76]	parentFdoExt→idleState is not protected by a lock	Incorrect result
Windows Kernel	Real data race example in [76]	gReferenceCount is updated without protection	Incorrect result
Trie benchmark	An example in [137]	The prefix match function reads the leaf field of the root object without acquiring a lock on the trie	Incorrect result

Table 4.1: Real examples of harmful asymmetric data races (20% of harmful data races are asymmetric).

Specifically, a library may have been written assuming a single-threaded environment, but later the requirements change to multithreading. This requires that all the threads acquire a lock before accessing shared state. Unfortunately, some corner cases are missed.

Asymmetric data races are likely harmful. Indeed, all of the ones shown in Table 4.1 that come from bug libraries have been confirmed as bugs in the libraries, and fixed in later releases of the software. In addition, the fact that the programmer protected one thread's accesses to the racy variables in a critical section suggests that these are important variables. The atomicity of the critical section accesses, as intended by the programmer, is broken through accesses from other threads; this is likely to be harmful.

4.2 PACMAN: Tolerating Asymmetric Data Races

Our goal is to tolerate asymmetric data races in production runs without the needing of training tests. This approach is complementary to conventional in-house data-race debugging. It is motivated by four facts. First, even after extensive testing, date race bugs appear in released

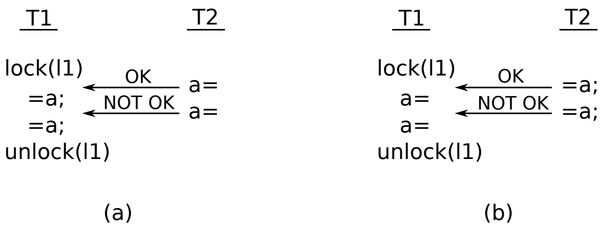


Figure 4.1: Examples of asymmetric data races where the *unsafe thread* can proceed (OK) or not (NOT OK).

code. Second, it often takes years between the time when a bug is detected in the field and when a fix is available from the vendor [179]. Third, for the fraction of asymmetric races caused by third party or (perhaps) legacy code, fixing the bug may not be a feasible option because the source code may be unavailable. Finally, the structure of these races already suggests a way to minimize their potential harm: prevent the *unsafe thread* from corrupting the state or reading inconsistent state while the *safe* one is in the critical section.

4.2.1 Overview of the Idea

We want to prevent *unsafe threads* from corrupting the state or reading inconsistent state while the *safe thread* is in the critical section. We must ensure that an access A from an *unsafe thread* that conflicts with an access inside the critical section is ordered in the same way with respect to all of the accesses in the critical section. As shown in Figure 4.1(a), the first write by T2 can proceed, but the second one has to be prevented until after the unlock is performed. Similarly, the first read by T2 in Figure 4.1(b) can proceed, but the second one has to wait until the unlock is performed.

The idea behind Pacman is to leverage the hardware cache coherence protocol in a multiprocessor to temporarily protect the variables that a thread is accessing in a critical section. The hardware performs two concurrent actions. One is to record the addresses of (a subset of) the variables that the *safe thread* is accessing while executing a critical section. In fact, to a large extent, we only need those addresses that can be observed by the cache coherence protocol, as we will explain below. The second action is to reject any requests from the *unsafe threads* that conflict with these variables, until the safe thread leaves the critical section.

For efficiency, Pacman does not record the addresses in a table. Instead, it uses hardware signatures (see Section 1.5). Moreover, to make the hardware as unintrusive as possible, the signatures are stored in a module called SigTable that is connected to the on-chip network and snoops all coherence transactions. Physically, the SigTable is associated with the network controller in a ring-based multiprocessor, or is distributed across the different directory modules in a directory-based multiprocessor. Since multiple cores may be executing critical sections concurrently, the SigTable stores as many signatures as critical sections are in progress.

The application code is unmodified, however, Pacman assumes that the critical section entry and exit points of *safe threads* are marked in the code with synchronization macros or libraries. Inside these macros or libraries, Pacman makes sure that there is a network access, implemented as part of the synchronization operation as we will see below. As a result, the Pacman module always knows when a core enters and exits a monitored critical section.

All previous works on tolerating data races are based on software systems [140] [137]. Hardware approaches mainly focus on detecting the data races [113] [33] [116]. This makes Pacman the first hardware approach to not only detect but also **tolerate asymmetric data races**.

Pacman addresses several shortcomings of the existing data race tolerance schemes. First, from a theoretical point of view, Pacman will not generate any new deadlocks or inconstancy behaviors that are not allowed by the original programs. Second, since we use hardware to check dynamic addresses on the fly, Pacman has more precise results than static alias analysis. Besides this, Pacman does not rely on hardware nor OS support for memory protection. Therefore, if the program uses fine-grained locks, Pacman would not cause any memory fragmentation. In fact, there is no need for compiler transformations or source code modifications.

Unlike existing hardware data race detection schemes, Pacman has several advantages. First, Pacman does not need additional hardware to support rollback and re-execution. Second, although it is a signature based system, false positive are typically not very significant, because Pacman only records the read/write addresses inside the critical section and usually the size of the critical section is small. Finally, there is no need to modifications on the cache coherence protocol or cache structure to support Pacman. We only need some small extensions on the cache coherence protocol to ensure that the module captures all the required information. The need of this extensions are discussed in Section 4.2.3.

In the following, we describe Pacman's operations in detail in three steps: basic Pacman protocol, interactions with cache, and the advanced Pacman protocol (to avoid deadlocks).

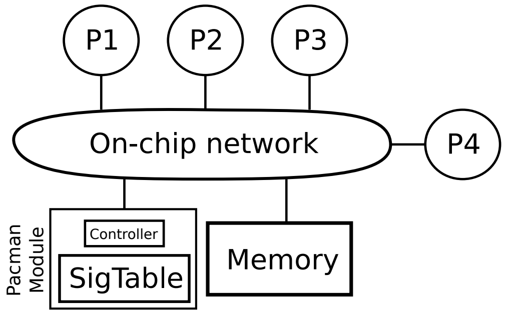


Figure 4.2: Overview architecture of Pacman.

4.2.2 Basic Pacman Protocol

As shown in Figure 4.2, the key component of Pacman is a SigTable and a controller, which are connected to the on-chip network. SigTable is a hardware component that stores the addresses accessed by each in-progress critical section, and prevent accesses by other cores to these addresses. Despite the SigTable can be implemented in a distributed way, here we describe a centralized module, for an instance implementation based on a ring on-chip network.

To detect asymmetric data races, the basic Pacman protocol needs a SigTable with several entries, each one with a CID and a signature. CID is the identifier of the core currently executing the critical section that owns the entry. The signature field contains the encoded addresses accessed in the critical section so far.

These two elements are enough to define the basic protocol, when a request arrives to the Pacman module:

1. **Lock acquire**: if the grab is performed successfully, Pacman allocates a new entry in the SigTable by assigning the CID to the requesting core ID, and inserting in the signature the address of the lock.

2. **Data access inside a critical section**: Pacman inserts the address of the data accesses in the corresponding signature of the SigTable entry.

3. **Data accesses from other threads**: the SigTable checks on every network access if the address is in the signatures of the SigTable. If there is a match, the request is Nacked to the requester, which will retry.

4. **Lock release**: Pacman deallocates the entry in the SigTable.

Nacks are often used in cache coherence protocols, to avoid having to buffer messages that cannot be processed immediately [70]. While they can cause traffic hot spots in pathological cases, the probability of an asymmetric race is low enough that there is no need to provide any contention management mechanism.

When a program uses nested locks, Pacman manages the situation by flattening, which means that the *signature* will store all the addresses inside the outermost lock and does not allocate a new entry to inner locks. To support this feature, Pacman needs a new element for each entry in SigTable, called *NestingLevel*. On a lock acquire, if the core does not own a SigTable entry yet, it sets the *NestingLevel* field to one, otherwise increments the value of the field. On a lock release, the Pacman module decrements the *NestingLevel* field, and, if it is zero, deallocates the entry in the SigTable.

With this simple protocol, Pacman isolates the *safe threads* from the *unsafe threads*, and it has negligible execution overhead for the *safe thread*.

4.2.3 Interaction with Cache

To minimize the hardware modifications to support Pacman, the module simply attaches to the on-chip network without modification of processor, cache coherent protocol, etc. However, in this way, Pacman can only monitor the events on the network, but some read/write operations which are not reported on the network may affect Pacman's correctness. Usually there are two kinds of read/write operations which cannot be detected in the network: write to a dirty data in the owner cache, and read of a clean data from the owner cache. We discuss the following cases when Pacman cannot monitor the events, assuming a basic MESI cache coherence protocol.

To illustrate the problem, we depict in Figure 4.3 two different situations. In the case of Figure 4.3(a), *T1* acquires *l1*, and then writes the *x* data, which misses in the cache. *l1* and *x* are inserted in the signature of SigTable entry, and any subsequent read or write from T2 will require a coherence transaction, which will be Nacked, as we can see in the figure.

Figure 4.3(b) shows the situation where *T1* reads *x*, which misses the cache, and Pacman records the addresses of *l1* and *x*. If *T2* reads *x*, there may be a coherence transaction (depending if the line is in the T2 local cache or not). If there is a transaction, the access will be Nacked, otherwise, it will not. The situation is ok in both cases because two reads never conflict. However, if *T2* writes *x*, the thread will be Nacked, as we observe in Figure 4.3(b).

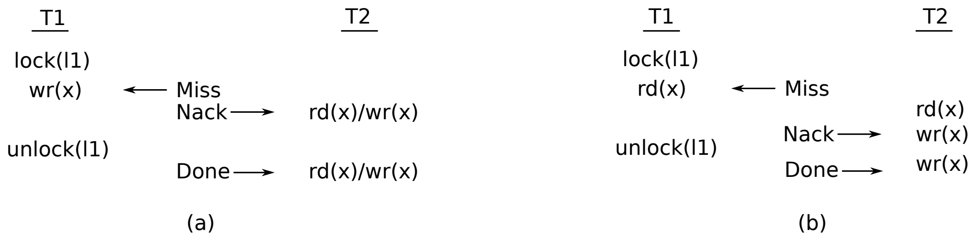
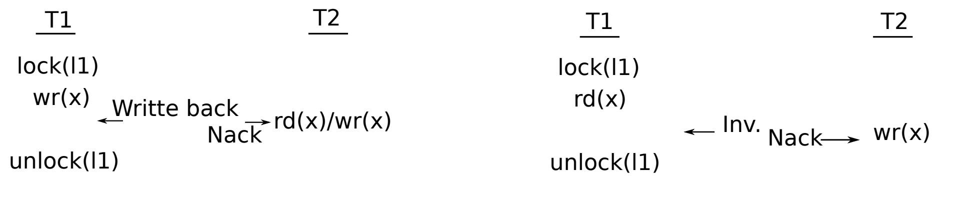


Figure 4.3: Examples to understand the Pacman's operation.



(a) x in Modified or Exclusive state before enter the critical section.

(b) x in Shared state before enter the critical section.

Figure 4.4: Examples to understand Cache State Prior to Entering the Critical Section.

These two cases are straightforward, and Pacman is able to manage them without architectural modifications. Next we will describe three cache coherence situations that are more complex to deal with: cache state prior to enter the critical section, cache replacements during the critical section, and synchronization operations.

Cache State Prior to Entering the Critical Section

When a thread enters in a critical section, some cache lines may be in a state that enables the core to silently access them. Specifically, there are two cases: when x is Dirty (or Exclusive) in T1's cache in Figures 4.4(a) and (b), and when x is Shared in T1's cache in Figure 4.4(b). In these cases, the Pacman module will not observe T1's access to x.

None of the two cases prevent Pacman from ensuring the atomicity of the critical section. Consider the case when x is in M or E state in T1. When T2 attempts to access the line and a miss is produced, the coherence protocol forces T1 to write back the line. When the Pacman module detects that a core with a SigTable entry writes back a line, it assumes that the core had accessed the line. Consequently, while allowing the line to be written back to memory, it inserts the line's address in the entry's Signature and Nacks the requesting (*unsafe*) core (hence ensuring critical section atomicity). No functional changes to the caches or coherence protocol is needed. If T1 had not accessed the data in the critical section, Pacman acts conservatively but not incorrectly.

In the second case, being *x* in Shared state in *T1*, if *T2* writes *x*, the hardware issues a coherence transaction that invalidates *T1's* copy. In this case, Pacman requires a simple hardware extension. Specifically, it requires that T1's cache informs, in its response to the invalidation, that indeed, it has invalidated a line. When the Pacman module detects that a core with a SigTable entry has invalidated a line, it assumes that the core has accessed the line in its critical section. Again, it may occur that this line has not been access in the critical section, but Pacman acts conservatively but in a correct way[2].

Supporting this change is simple. In a directory-based protocol, when a cache invalidates a line, it must set a bit in the invalidation acknowledgement returned to the directory. In a snoopy-based protocol, the cache that invalidates the line must set a bit in the network that is visible to the Pacman module.

Cache Replacements During the Critical Section

Consider the case when a core is executing a critical section and its cache evicts a line that was in the cache before the core entered the critical section. Such line is not in the signature, but it must be conservatively put there as the core may have accessed it silently during the critical section.

There are two kinds of replacements: if the cache line is dirty (M state), the line is written back to memory, so the Pacman module detects this transaction. However, if the replaced line is clean, the Pacman module does not have any notification of the event. Therefore, another modification of Pacman is required: to send a notification (including the address line) when a cache replaces a clean line in a critical section, with the aim of including it in the corresponding SigTable entry. This modification can be implemented easily. Specifically, a new counter named *Mode* is added to the controller of the last level private cache. When *Mode* is not zero, the cache is in notification mode, and it sends a notification each time that replaces a clean line. Every successful lock acquire increments the counter and every unlock decrements it. This ensures that, in nested critical sections, the cache remains in notification mode throughout the outermost critical section.

[2]In all of these cases, a Nacked write has already invalidated the line from all the caches. This can hurt performance slightly if caches have to re-access the data. However, this occurs only once.

Synchronization Operations

Every acquire and release operation should be notified to the Pacman module to allocate or release a new entry in the SigTable. But when the lock is in a line in M or E state, this does not happen. To solve this problem, we propose to send an extra notification in each successful acquire or release operation that do not need a network access. Another alternative would be to change the lock macros or libraries to add an explicit uncached write inside them.

4.2.4 Advanced Pacman Protocol to Avoid Deadlocks and Stalls

To deal with more involved cases, we describe possible deadlock situations and the mechanism that Pacman uses to avoid them.

Stalls and Deadlocks

The potential deadlock situations in a Pacman basic protocol are three:

1. Some race bugs where all the threads synchronize.
2. False sharing.
3. False positives.

Figure 4.5 shows two examples of the first situation. In Figure 4.5(a), T1 and T2 acquire two different locks (*l1* and *l2*) and enter in their respective critical sections. As both are *safe threads*, both are protected against external accesses to the critical data, and both threads accessing at the same data (a1 and a2) with a timing that produces the stall of both threads. Figure 4.5(b) shows two threads (T1 and T2) acquiring two different locks (*l1* and *l2*) accessing the same variable (a2). T2 successes accessing a2 and T1 gets Nacked, but T2 tries to acquire *l1* (acquired by T1) and gets also stalled.

The second possible deadlock situation is false sharing. For example, this would be the case of Figure 4.5(b), if T1, instead of accessing a2, access to another variable that share the cache line with a2.

The third situation of deadlock is due to the false positives in signatures due to aliasing (see Section 1.5) or due to the cache state prior to entering the critical section (see Section 4.2.3).

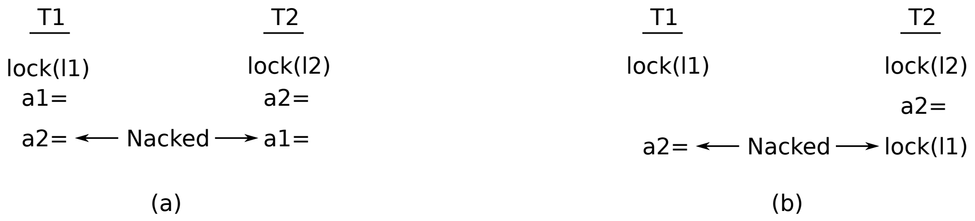


Figure 4.5: Examples of data race bugs which lead to deadlock.

Mechanism to Avoid Deadlocks

The mechanism used by Pacman to avoid deadlocks are described here. To handle deadlocks, we add two new fields to each entry of the SigTable:

1. *Stall_index*: tells if the thread that owns the entry is being Nacked. Specifically, *Stall_index* stall the index of the SigTable entry that sends Nacks to the owner thread.
2. *Lock_Acquire?*: indicates if the thread that owns the entry is being Nacked while trying to acquire a lock (if *Stall_index* is not null).

The algorithm to avoid the deadlock is the following:

– When a core C_i is Nacked by the core corresponding to the j entry of the SigTable, the Pacman module checks if C_i also has a entry in the SigTable. If so, it sets the value of *Stall_index* to j in the entry corresponding to the C_i core.

– It sets also *Lock_Acquire?* bit in case C_i is trying to acquire a lock.

– The Pacman module follows the $Stall\_index$ pointer by checking entry j in the SigTable and reading its own $Stall\_index$.

– If, by following the $Stall\_index$ pointers in this way, the hardware ends up in entry i, it has detected a cycle.

Once the deadlock is detected, the hardware needs to decide which thread among those in the cycle is allowed to perform one access without being Nacked. A simple approach is to pick one of the threads that holds locks requested by other threads (such as T2 in Figure 4.5(b)). Such threads are detected from the *Lock_acquire*? bit of other entries, and they need

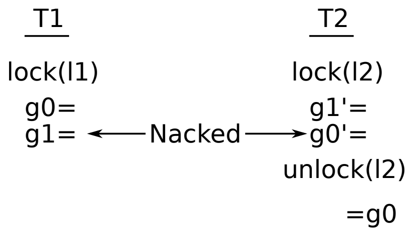


Figure 4.6: Breaking atomicity due to false sharing.

to make progress to break the cycle. If there is no such thread, the hardware picks one thread at random. The next time that the Pacman module detects a request from the picked thread, it does not Nack it.

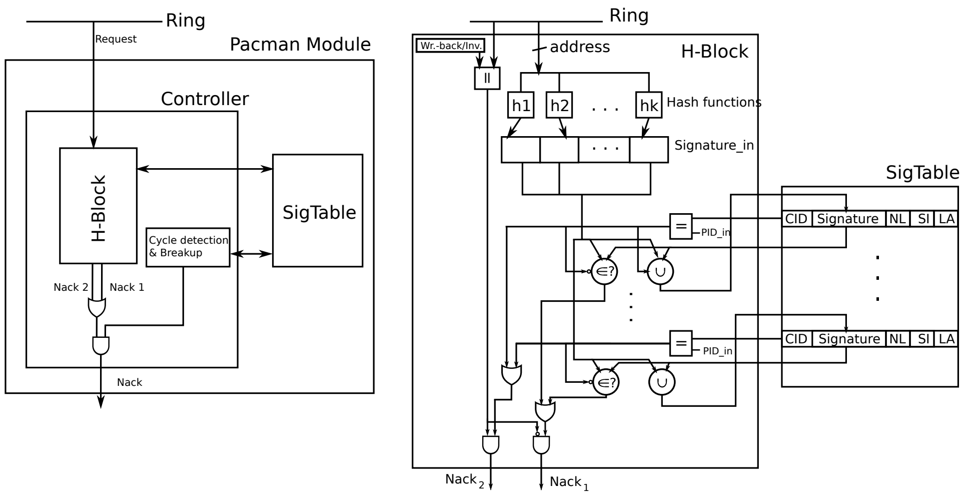
Breaking Atomicity of Critical Sections

With the algorithm described in the previous section, Pacman immediately finds and breaks any deadlock — unless it was already present in the original application. However, by letting one stalled thread complete one access, it can conceivably break the atomicity of a critical section. To understand the problem, we consider each of the three sources of deadlock listed in Section 4.2.4.

In the first case (all the threads synchronize), Pacman can potentially break the atomicity of one of the critical sections. While Pacman could be designed to break only the atomicity of *unsafe threads*, such approach would not work for all the race bugs. An example is when T1 in Figure 4.5(b) is the *unsafe thread*. Overall, given the very low probability of breaking atomicity in this way, we do not attempt to avoid it.

In the second case (false sharing), atomicity can potentially be broken unless special care is taken. To see why, consider Figure 4.6, which is a slightly modified version of Figure 4.5(a). In this example, variables g0 and g0’ share the same cache line, while g1 and g1’ share another line. Because of false sharing, threads T1 and T2 deadlock. By breaking the deadlock through letting T2 read g0’, Pacman is allowing the line to go to T2’s cache. Right after the end of T2 critical section, T2 could attempt to silently access g0 from its cache, which could break T1’s atomicity.

To prevent this case from occurring, we could augment Pacman so that, when Pacman lets one access break a deadlock, it marks it as non-cacheable. The requesting core would be allowed to use (read or write) the word, but its cache would not be allowed to keep the line.



(a) Pacman Module. (b) SigTable and H-Block.

Figure 4.7: Pacman Implementation.

As a result, accesses to other words would miss in the cache. This extension would avoid breaking atomicity when false sharing occurs between words. However, a more elaborated solution would be needed when false sharing occurs between bytes of the same word. Given the very low probability of breaking atomicity due to false sharing, Pacman does not include this support (it can not handle this case).

In the third case (false positives), letting one thread proceed does not break the atomicity of any critical section.

4.3 Implementation Issues

The Pacman module is a hardware module connected to the on chip network (see Figure 4.7(a)). It comprises the SigTable and its controller. The controller is composed of one simple hash block (H-Block) and the cycle detection & breakup module. The latter chases the *Stall_index* links as described in Section 4.2.4 to detect and break deadlocks.

Figure 4.7(b) shows the H-Block and the SigTable. The implementation is optimized to reduce the complexity of the controller. In case of an ordinary request in the network, the H-Block takes the address of the incoming request and insert it into an empty signature using

CID	N bits
Signature	1k bits
NestingLevel (NL)	5 bits
Stall_index (SI)	N bits
Lock_acquire? (LA)	1 bit

Table 4.2: Size of the SigTable's fields.

a parallel Bloom filter (*Signature_in* in Figure 4.7(b)). The hash-encoded address (stored in *Signature_in*) is then checked for membership in valid SigTable entries from other cores ($\in$ or check operation in Figure 4.7(b)). Overall, in case of an ordinary request in the network, the H-Block's operations can be performed in 2-3 cycles and are hidden under the first half of the network transaction.

In the second half of the network transaction, when the caches have finished snooping, the network may receive a write back or invalidation response (Section 4.2.3). In this case, the H-Block checks if the core ID that writes back or is invalidated has a SigTable entry. If so, it bit-wise ORs the hashed address with the correct signature and raises the signal Nack2 . In this case, the H-Block's operation takes 1-2 cycles.

If Nack1 or Nack2 are raised and the cycle detection and breakup module does not prevent it, a Nack signal is returned on the network. All of the operations of the Pacman module except for cycle detection are simple enough to be overlapped with the network transaction. In a directory protocol, they overlap with directory module accesses. The cycle detection may take over 10 cycles, which is acceptable since it is done in background.

In our current implementation, the sizes of the SigTable's fields of Figure 4.7(b) are shown in Table 4.2. The size of CID and *Stall_index* depend on how many threads may be monitored at a time (the maximum number of threads is 2^N). For *Signature*, we found that, with 1,024 bits, false positives are typically less than 1%. For NestingLevel, we allocate 5 bits, which is enough for all the benchmarks used in the evaluation.

Finally, the Pacman module is enabled and disabled by the *Pacman on* and *Pacman off* commands, respectively. These can be implemented as writes to memory-mapped registers. These commands can be used to exclude the program regions that are serial or otherwise uninteresting. These two commands would be mainly used to exclude serial regions of programs and shared libraries. This way, overheads and signature pollution are reduced when Pacman is not necessary.

4.3.1 Other Issues

Current systems usually support two functionalities that we did not mention up to now: multithreading and OS thread migration. Multithreading is when a core supports multiple threads at the same time, and thread migration is when the OS migrates one thread from one core to another.

The previous discussion of Pacman assumes a single threaded core without virtualization support. In this section, we discuss how Pacman supports these two functionalities in a simple way. Furthermore, we also show a distributed version of the Pacman module.

Virtualization: Pre-emption and Thread Migration Support

While executing a critical section, a thread can be pre-empted and even migrated to another core. In an advanced design that requires OS support, we would like that (i) while a thread is pre-empted in a critical section, we keep protecting its critical section, and (ii) when it resumes in a potentially different core, we keep storing its accesses in the same signature. To support this, when the OS pre-empts a thread from core i, it checks the SigTable for an entry with CID equal to i. If it finds one, it changes its CID field. Specifically, if the thread does not run anywhere, it sets the CID field to a special code (e.g., OUT); if it finally runs on core j, then it sets the CID field to j.

With this scheme, if a thread gets pre-empted and not running, it still has its critical section protected from asymmetric races. Indeed, its SigTable entry is still valid and coherence messages are checked against its signature. The checks may result in sending Nacks. Then, when the thread is scheduled on a different core, its accesses are still stored into the same old signature.

This approach is efficient, since there is no copying or saving/restoring of SigTable entries. Moreover, the hardware is kept simple, since it always does the same thing: store accesses from core i into the SigTable entry tagged with CID i. $Stall\_index$ does not get stale, since it contains a table index. If the program has more threads than cores, there may be several SigTable entries with a CID equal to OUT. In addition, at a given time, the SigTable entries may belong to threads from several different programs. Pacman works correctly because it uses physical addresses.

There is an issue with the cache state left behind by a thread that migrates while executing a critical section. Recall from Section 4.2.3 that the thread may have entered the critical section with a cache state that it is later accessed while in the critical section without notifying the Pacman module. We showed that Pacman (conservatively) captures this information at cache replacements or at write-backs/invalidations triggered by other core. However, if we now migrate the thread, we cannot capture such events.

To keep the design simple, we accept this limitation. This means that Pacman misses the few cases listed in Section 4.2.3 for threads that migrate while in a critical section. A more aggressive approach would be to write back to memory all the dirty cache lines at the time the thread migrates while in a critical section. The addresses of these writebacks would be put in the signature. A more drastic approach would be not to allow migration during critical section execution. Overall, since critical sections are typically small, migration during their execution is rare and does not justify additional actions. Like all data-race handling techniques, Pacman is a best-effort approach (the probability of not tolerating an asymmetric data race is very low).

Multithreading Support

Multithreaded cores have multiple hardware contexts and run multiple threads at a time. It is possible that different threads executing on different contexts of the same core concurrently execute different critical sections. In this environment, Pacman requires an extension where the messages sent by cores to the SigTable include both the core ID and the hardware context ID within the core. Similarly, SigTable entries have both a CID and a ContextID field.

The cache-state issues of Section 4.2.3 are handled conservatively. If multiple contexts in a core are concurrently executing critical sections, any writeback, invalidation, or replacement that needs to insert an address in a signature, lead to insert it in all the SigTable entries owned by that core. Since the SigTable is connected to the network, it can only observe data sharing across cores, not across contexts in a core. Consequently, for Pacman to tolerate races as described, a program can only use one context per core — although multiple programs can use the multiple contexts of a core. To allow a program to use multiple contexts in a core, bigger changes would be needed, such as stalling all the other threads in the core when one thread is executing a critical section.

Extensions for a Distributed Pacman Module

The discussion so far assumed a centralized Pacman module, which is reasonable for a snoopy protocol. To use Pacman in a system with a directory-based protocol, we need to distribute the Pacman module across the different directory modules. Since such a design is outside our scope, we only outline it briefly. Like the directory, the module naturally lends itself to a distributed environment, with partitions based on address ranges. Consequently, each directory module has an associated Pacman module, which is in charge of the range of physical addresses assigned to the local directory module. When a thread enters a critical section, the hardware allocates an entry for the core in the SigTable of all the Pacman modules; when it exits it, all the entries are deallocated. When a thread misses on an address, the request naturally reaches the home directory of that address. There, the address is checked against the entries in the local Pacman module using the usual algorithm. The Pacman modules in the other directory modules are not checked.

4.4 Evaluation

4.4.1 Experimental Setup

To evaluate the potential and performance of Pacman, we model Pacman by using the software framework for dynamic library instrumentation Pin [81] connected to a cycle-by-cycle execution driven architecture simulator based on SESC [142]. The simulator models a chip multiprocessor of 4 or 8 cores, configured by default by the parameters in Table 4.3. The cores are two-issue, in-order, and overlap memory accesses with instruction execution. Each core has a private cache hierarchy kept coherent by a basic MESI coherence protocol on an on-chip bus. The bus is connected to the Pacman module and to off-chip main memory. Unless otherwise indicated, the sizes of the fields in a SigTable entry are those shown in Table 4.4. To generate a signature, Pacman uses eight 128-bit Bloom filters in parallel using the H3 hash function from [147], for a total of 1,024 bits per signature.

For sensitivity analysis, we consider two cache hierarchy models, namely one where each core only has an L1 cache, and one where it has both a private L1 and a private L2. The first model puts more pressure on Pacman.

We evaluate Pacman with all the fourteen SPLASH-2 applications, the twelve PARSEC applications that support pthreads, the "Sphinx3" speech recognition software [150], and "Apache 2.2.3". The SPLASH-2 codes use their default inputs, while the PARSEC codes use the *simmedium* inputs. For "Sphinx3", we use the test input provided, which executes over 500 million instructions, while for "Apache", we set up clients that keep sending requests to the server, so that the server executes around 40 million instructions.

Architecture	CMP with 4 or 8 cores
Core type	2-issue, in-order, 1Ghz
Private L1 cache	32Kbytes, 4-way assoc., 64byte lines
Private L2 cache	512Kbytes, 8-way assoc., 64byte lines
L1 hit latency	2 cycles round trip
L2 hit latency	8 cycles round trip
L2 miss latency	30 cycles round trip to other L2s
L2 miss latency	250 cycles round trip to other Memory
Coherence protocol	Snoopy basic-MESI on 64byte bus
SigTable parameters	From Table 4.4
Cycle detection latency	4-14 cycles
H-Block latency (with a normal request)	2 cycles
H-Block latency (with a Write-back or Nack)	2 cycles

Table 4.3: Default architecture parameters.

SigTable entries	8 (Max.)
Signature size	1024 bits
Signature structure	8 128-bit Bloom filters with H3
CID	2 bits
Stall_index	2 bits
NestingLevel	5 bits
Lock_acquire	1 bit

Table 4.4: SigTable parameters.

For the evaluation, we slightly modify the "Canneal" and "Ferret" applications. At the beginning of "Canneal", a thread uses a critical section to initialize a large memory space (even though there is no other active thread at that time). Consequently, we turn off Pacman during that time. In "Ferret", each thread initializes a random number generator within a critical section. Since only the seed is a shared variable, we move the local-variable accesses in the random number generator initialization routine outside of the critical section. If we did not do these changes, the statistics on critical section sizes (Section 4.4.2) would be biased. In addition, for "Ferret", if all the local-variable addresses were inserted into the signature, this could potentially induce, through address aliasing in the signatures, false positive conflicts with other threads, and unnecessarily stall them.

In the rest of this section, we characterize the critical sections, evaluate the overheads of Pacman, and examine the asymmetric data races discovered. On PARSEC benchmarks, we use a *pthreads* configuration to run them as multithread applications. The critical sections are also defined by a pair of *pthread_mutex_lock* and *pthread_mutex_unlock* in the pthreads library. We use all twelve benchmarks supported in the *pthreads* configuration in PARSEC.

4.4.2 Characterization

Table 4.5 characterizes the critical sections in all of the 28 applications on the 4-core multicore processor. Column 3 lists the number of dynamic critical sections in each program. Column 4 shows the percentage of the dynamic instructions in the programs that are inside the critical sections. We see that all programs except "Sphinx3" execute less than 1% of their instructions in critical sections. The percentage in "Sphinx3" is 3.5%. Columns 5 and 6 show the average and maximum number, respectively, of instructions executed per critical section. We see that the applications tend to have modest-sized critical sections. Most applications execute less than 100 instructions per critical section on average. The maximum number of instructions in a critical section reaches nearly 7,000 in "Vips". Columns 7-8 list the average number of reads and writes per critical section.

Columns 9,10 and 11 correspond to results obtained with the architecture with only the L1 caches. They show, per critical section, the average number of clean line replacements, and the average and maximum number of line addresses included in the signature. We see that the average number of clean replacements per critical section is close to zero. This means that this effect is minor. The average number of line addresses included in a signature per critical section is typically less than 10 and, except for a few cases, the maximum number is not much higher. These numbers suggest that the probability of false positives in the signatures is low. Note that for the architecture with both L1 and L2 private caches, these numbers will be smaller, because caches keep more state in this case.

The last column shows the maximum nesting level of critical sections. A value more than one means that the application has nested locks. The value one indicates that the benchmark has no nested locks and the value zero means no critical section in the benchmark. Only "Radiosity" and "Vips", which have a recursive structure, have significantly deeper levels.

Overall, given the typical sizes and properties of the critical sections observed, we believe that a simple solution for asymmetric race detection is enough. Pacman provides such a simple solution.

4.4.3 Overheads

There are two sources of execution overhead in Pacman. The first one is that some cores receive Nacks and have to retry. The second one is additional network traffic created by three event types: a notification message in a clean replacement inside a critical section, a retry

1	2	3	4	5	6	7	8	9	10	11	12
Category	Application	# Dynamic CS	CS insts (%)	#insts per CS	Max insts in CS	#reads per CS	#writes per CS	#disps per CS	#sig addrs per CS	Max # sig addrs in CS	Max CS nesting level
SPLASH-2 Kernels	cholesky	6,957	0.0	30.3	161	10.7	4.7	0.0	6.4	11	1
	fft	32	0.3	33.9	47	11.9	10.5	0.1	5.8	7	1
	lu/contiguous	272	0.0	36.1	47	12.6	10.7	0.0	6.0	7	1
	lu/non_cont.	80	0.0	35.2	47	12.4	10.5	0.1	5.8	8	1
	radix	78	0.0	26.1	47	9.4	8.4	0.0	5.9	7	1
SPLASH-2 Apps	barnes	68,938	0.4	118.1	1,898	40.1	29.3	0.0	11.9	56	1
	fmm	44,622	0.2	142.1	252	54.7	27.9	0.0	13.4	21	1
	ocean/cont.	4,432	0.0	31.5	45	11.8	9.6	0.0	6.8	9	1
	ocean/non_cont.	4,312	0.0	30.9	45	11.8	9.5	0.0	5.9	7	1
	radiosity	273,087	0.9	18.2	1,226	8.7	5.9	0.0	5.6	89	5
	raytrace	95,475	0.3	29.3	6,661	7.5	5.8	0.0	6.0	343	1
	volrend	72,524	0.0	12.1	50	5.0	3.0	0.0	4.9	8	1
	water-nsquared	6,292	0.0	50.3	51	34.4	12.4	0.0	17.0	18	1
	water-spatial	157	0.0	23.8	47	9.6	7.0	0.0	6.0	9	1
PARSEC Kernels	canneal	4	0.0	7	10	2.5	3.5	0.0	3.3	4	1
	dedup	17,932	0.1	315.9	802	121.2	67.9	0.1	14.4	33	1
	streamcluster	52,128	0.0	21.0	32	7.1	4.8	0.0	3.2	5	1
PARSEC Apps	blackscholes	0	-	-	-	-	-	-	-	-	-
	bodytrack	8,273	0.0	37.0	1,228	15.6	11.1	0.0	6.9	34	1
	facesim	7,921	0.0	37.0	154	18.0	9.9	0.0	5.4	11	2
	ferret	733	0.0	19.2	44	5.4	7.2	0.0	5.0	9	2
	fluidanimate	2,113,870	0.7	15.9	32	10.2	4.1	0.0	8.0	10	1
	raytrace	73	0.0	7.8	31	2.6	2.3	0.0	2.2	6	1
	swaptions	0	-	-	-	-	-	-	-	-	-
	vips	14,056	0.0	49.0	6,723	18.6	11.8	0.0	8.0	106	23
	x264	4,071	0.0	10.6	39	5.9	1.7	0.0	4.0	6	1
Other Apps	Apache	8,301	0.4	24.4	40	9.7	5.3	0.0	5.6	8	1
	Sphinx3	94,382	3.5	208.5	2,946	86.7	29.1	0.1	6.0	243	2

Table 4.5: Characteristics of the critical sections (CS) in the applications.

after a Nack, and the extra message in a successful lock acquire or release that hits on a cache line that is in Dirty or Exclusive state (see Section 4.2.3).

Table 4.6 quantifies these effects for each application. Columns 3 to 8 show the total number of Nacks observed during the execution of the application. For each application, we performed 3 to 5 runs, and show the maximum number of Nacks seen in any individual run. The data corresponds to the architecture with L1 caches only, which is the worst case. Columns 3 to 5 correspond to 4-core runs, while Columns 6 to 8 correspond to 8-core runs. For "Apache", since the server automatically sets the number of threads to a number larger than 8, we put the data under the 8-thread columns. In each group of three columns, the first one shows the Nacks observed due to true conflicts (i.e., two threads access the same variable), the second one shows the Nacks due to true conflicts or false sharing, and the last one shows the Nacks due to true conflicts, false sharing, or false positives.

The number of Nacks is very small. Only "FMM" and "Bodytrack" exhibit Nacks due to true conflicts. Each of them has one Nack. False sharing and false positives increase the number of Nacks. The highest number is 32 for "Radiosity". This is negligible compared to the 454M dynamic instructions executed by "Radiosity". Overall, the impact of any core stall due to Nacks is insignificant.

Columns 9 to 10 show the percentage increase in network traffic due to the three effects listed above. Column 9 applies to the architecture with L1 private caches only, while Column 10 applies to the one with L1 and L2 private caches. The data shows that the increase in traffic is very small. In the worst case, the increase is 1.5% for the case of L1 private caches and 2.4% for the case of L1 and L2 private caches. These low numbers result from the fact that critical sections have a modest size and account for a small fraction of the execution time. Overall, the impact of this extra traffic is negligible.

Column 11 shows the number of successful lock acquires and releases that hit on a cache line that is in Dirty or Exclusive state and, therefore, introduce an additional bus access. The data corresponds to the architecture with both L1 and L2 caches. The column gives the number of such events as a percentage of dynamic instructions. We see that, typically, such number is negligible. In the worst case, we have 0.05 such events per 100 instructions. Therefore, the impact of such events is insignificant.

Finally, Figure 4.8 shows the increase in the execution time of the applications due to all of the Pacman overheads combined. The data is shown as a percentage of the original execution time of the applications for 1, 4 and 8 threads. There is a data point for each program, and a

1	2	3	4	5	6	7	8	9	10	11
		Number of Nacks (L1 only, 4 threads)			Number of Nacks (L1 only, 8 threads)			Increase in traffic with	Increase in traffic with	Sync hits per dyn
Category	Application	True	True+FS	True+FS+FP	True	True+FS	True+FS+FP	L1 only (%)	L1+L2 (%)	inst (%)
SPLASH-2 Kernels	cholesky	0	0	0	0	0	0	0.0	0.0	0.00
	fft	0	0	0	0	0	0	0.0	0.0	0.00
	lu/contiguous	0	0	0	0	0	0	0.0	0.0	0.00
	lu/non_cont.	0	0	0	0	0	0	0.0	0.0	0.00
	radix	0	0	0	0	0	0	0.0	0.0	0.00
SPLASH-2 Apps	barnes	0	2	4	0	2	4	0.0	0.3	0.01
	fmm	1	1	1	1	1	1	0.0	0.1	0.00
	ocean/contiguous	0	0	0	0	0	0	0.0	0.0	0.00
	ocean/non_cont.	0	0	0	0	0	0	0.0	0.0	0.00
	radiosity	0	13	15	0	28	32	1.0	1.4	0.04
	raytrace	0	0	4	0	0	6	0.0	0.1	0.01
	volrend	0	0	0	0	0	0	0.0	0.1	0.00
	water-nsquared	0	0	0	0	0	0	0.0	0.1	0.00
	water-spatial	0	0	0	0	0	0	0.0	0.0	0.00
PARSEC Kernels	canneal	0	0	0	0	0	0	0.0	0.0	0.00
	dedup	0	0	0	0	2	2	0.1	0.2	0.00
	streamcluster	0	0	0	0	0	0	0.0	0.0	0.00
PARSEC Apps	blackscholes	0	0	0	0	0	0	0.0	0.0	-
	bodytrack	1	1	2	1	1	2	0.0	0.0	0.00
	facesim	0	0	0	0	0	0	0.0	0.0	0.00
	ferret	0	0	0	0	0	0	0.0	0.0	0.00
	fluidanimate	0	0	0	0	0	0	1.5	2.4	0.05
	raytrace	0	0	0	0	0	0	0.0	0.0	0.00
	swaptions	0	0	0	0	0	0	0.0	0.0	-
	vips	0	0	2	0	0	3	0.0	0.0	0.00
	x264	0	0	0	0	0	0	0.0	0.0	0.00
Other Apps	Apache	-	-	-	0	3	8	0.3	0.5	0.02
	Sphinx3	0	4	6	0	10	14	0.8	1.1	0.02

Table 4.6: Quantification of the overheads.

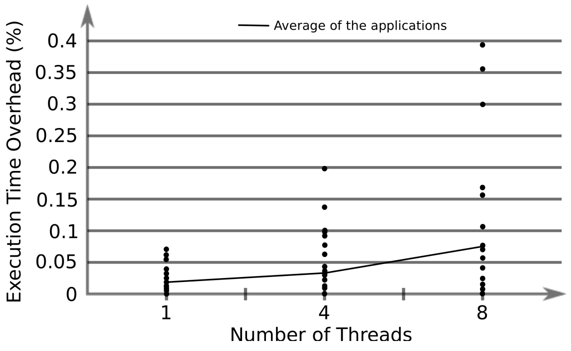


Figure 4.8: Execution time overhead of Pacman.

line for the average of all of them. The figure shows that, even for 8 threads, the maximum overhead in any application is only 0.4%, while the average is only 0.07%. The figure also shows that, for most applications, the overhead increases slowly with the number of threads. The overhead for 1 thread is due to the extra bus accesses in synchronizations. Overall, the execution time overhead of Pacman is negligible.

4.4.4 Handling Bugs

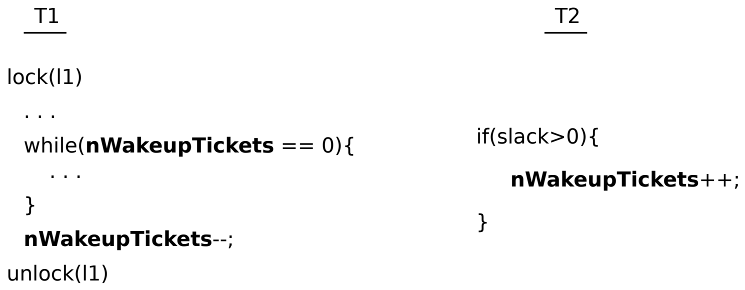
The Pacman module found two unreported bugs in the PARSEC benchmarks and it was able to tolerate all asymmetric data races in an artificial bug-intensive test.

Unreported Data Race Bugs

Since SPLASH-2 and PARSEC are widely used benchmarks, usually they are well synchronized and Pacman cannot find asymmetric races in most applications. However, in PARSEC, we found two asymmetric races, one in "Bodytrack" application (Figure 4.9) and another one in "FMM" application (Figure 4.10).

In the case of "Bodytrack" (Figure 4.9), T1 accesses nWakeupTickets before and after a while loop inside a critical section and T2 accesses nWakeupTickets outside the critical section. We suggest that it is not necessary to put nWakeupTickets inside the critical section.

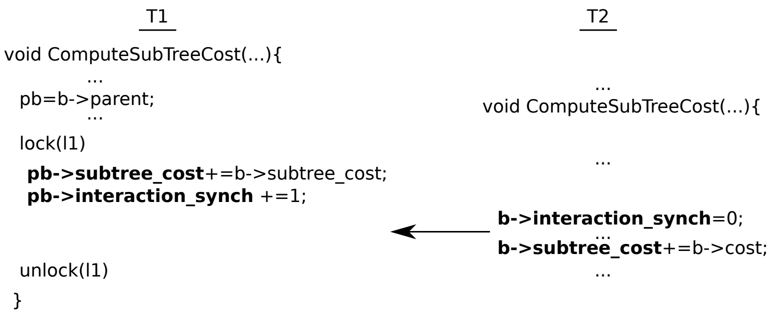
The asymmetric race in "FMM" is shown in Figure 4.10. It happens in subroutine ComputeSubTreeCosts, where multiple threads are accessing a tree structure. When two threads T1 and T2 are concurrently executing the subroutine, it may be that the two point to the same



```
T1
lock(l1)
  . . .
  while(nWakeupTickets == 0){
     . . .
  }
  nWakeupTickets--;
unlock(l1)
```

```
T2
if(slack>0){
    nWakeupTickets++;
}
```

Figure 4.9: An asymmetric race in "Bodytrack" benchmark.



```
T1
void ComputeSubTreeCost(...){
        ...
 pb=b->parent;
        ...
 lock(l1)
  pb->subtree_cost+=b->subtree_cost;
  pb->interaction_synch +=1;


 unlock(l1)
}
```

```
T2
          ...
void ComputeSubTreeCost(...){

          ...

 b->interaction_synch=0;
          ...
 b->subtree_cost+=b->cost;
          ...
```

Figure 4.10: An asymmetric race in "FMM" benchmark.

node from two different places (pb in T1 is the same as b in T2), and an asymmetric data race may happen.

Artificial Inserted Bugs

We also modified some SPLASH-2 benchmarks to have intentional asymmetric race collisions on their shared variables. We assume that all original threads in SPLASH-2 are *safe threads*. We concurrently create an extra *unsafe thread* that continuously write random values to the shared variables without any locks. These accesses cause asymmetric race problems during the execution. In the tests on Pacman simulation, we checked that Pacman detects and tolerates all asymmetric race cases introduced.

4.5 Related Work

4.5.1 Software Proposals for Asymmetric Races

To put our work in perspective, we describe in detail two existing proposals to tolerate asymmetric data races, namely, ToleRace [140] and ISOLATOR [137]. Both schemes are software-only (i.e., no hardware support is provided). We then summarize Pacman's advantages over them.

In ToleRace, when a *safe thread* T_s enters a critical section, it makes two copies in software of all the shared variables in the critical section. Let us call the original variables V and the two copies V' and V". The *safe thread* then executes the critical section reading and writing V'. In the meantime, any *unsafe thread* T_u can access the original variables V. When T_s completes the critical section, it compares V and V". Based on whether V and V" are the same and on a knowledge of the access pattern interleaving of T_s and T_u , the *safe thread* makes one of three choices: *(i)* when T_u's execution can be serialized before T_s's, it copies in software V' to V, *(ii)* when T_s's execution can be serialized before T_u's, it leaves V as it is, and *(iii)* when the execution of T_u and T_s cannot be serialized in any way, it interrupts the program. In cases *(i)* and *(ii)*, the race has been tolerated; in case *(iii)* the race induces a sequentially inconsistent execution and, therefore, ToleRace is unable to handle it.

ToleRace has several shortcomings. First, a race type of case *(iii)* cannot be handled adequately: leaving version V or V' produces an inconsistent execution (a detailed example is described in [137]). Second, when the critical section contains multiple variables and accesses, the analysis of what is the race case may become complicated. Third, analysis of access patterns is either conservative (if static) or slow (if dynamic). Finally, comparisons and copies are slow and race-prone.

ISOLATOR [137] takes a different approach. When a *safe thread* T_s enters a critical section, it makes a copy in software of the pages that contain the shared variables that will be accessed in the critical section (Shadow Pages). Then, it changes the protection bits of the original pages to make them inaccessible. T_s operates on the shadow pages. If an *unsafe thread* T_u accesses the original pages, it gets an exception and gets de-scheduled. When T_s leaves the critical section, it copies the shadow pages back to the original pages and unprotects the latter.

ISOLATOR has the advantage of always producing consistent executions. In addition, thanks to an optimization, the number of page copies can be reduced. However, it has several shortcomings. The first one is the substantial compiler and operating system (OS) support (or code re-writing by the user) required to place variables in the correct pages and adapt to changing access patterns in the program. To apply ISOLATOR to PARSEC, we would have to rewrite the code and change the variable allocations significantly. A second shortcoming is that, if such rewriting is not provided, ISOLATOR will often need to copy large amounts of data at critical section entries and exits. For example, such data copying is the main reason why ISOLATOR reports up to 8x overhead for the microbenchmarks in [137]. Finally, ISOLATOR is prone to deadlocks and livelocks due to false sharing at page level — e.g., assume that the *unsafe thread* T_u gets de-scheduled and then T_s attempts to access a variable in a page that T_u has protected. Moreover, the timeout-based mechanism that is used to detect such deadlocks is very slow.

Overall, we conclude that neither ToleRace nor ISOLATOR provides the desired solution to handle asymmetric races.

4.5.2 Other Related Work

Pacman is related to Transactional Memory (TM) (Section 1.4) in that it presents a concept analogous to strong atomicity [101] between a transaction and a non-transactional access. However, Pacman operates on lock-based code. Moreover, compared to HTM, Pacman does not need speculation, rollback, timestamp support, or version management. Even to detect inter-thread conflicts, Pacman cannot leverage HTM's tagging of cache lines: since Pacman is non-speculative, data can overflow into memory. Hence, Pacman needs to keep a SigTable in memory. Compared to STM, Pacman does not need to analyze the code.

Pacman is also related to hardware-based mechanisms for fine-grain memory protection, such as UFO [15] and iWatcher [187]. In UFO, each memory line has some bits that specify protection information. Such bits travel with the line to caches. It is possible to support Pacman-like functionality with UFO. However, UFO is substantially more intrusive, as it requires maintaining these distributed bits and building exception handlers for them. iWatcher is similar although it targets single core processors.

Moreover, Pacman is also related to the many software or hardware schemes that detect and avoid atomicity violations, such as AVIO [92], AtomAid [94], AtomTracker [112], or LifeTx [185]. While Pacman focuses on avoiding races rather than atomicity violations, its hardware is effectively being used to keep atomicity, albeit for only user-defined critical sections. As a result of the latter, Pacman needs no training runs. Finally, there are some software-only schemes to tolerate races and bugs, such as Rx [133] or Frost [172]. Such techniques, while effective, have substantially higher overheads. We find Pacman to have negligible overhead.

Pacman differ from the previous proposals in that is the first hardware approach that detects asymmetric races, it is not based on data replication to maintain correctness in critical sections, and introduce a low overhead hardware solution based on signatures. Pacman also solve the inconsistency problems of ToleRace and the deadlocks of ISOLATOR.

4.6 Conclusions

In this chapter we propose Pacman, the first scheme designed to tolerate asymmetric data races in production runs with negligible execution overhead. Pacman leverages cache coherence hardware to temporarily protect the variables that a thread accesses in a critical section. Unlike the previous, software-based schemes, Pacman induces negligible slowdown, needs no compiler or (in the base line design) OS support, and requires no application source code changes. Moreover, its hardware is unintrusive since it is concentrated in a module in the network, rather than in the cores. We evaluated Pacman for SPLASH-2, PARSEC, "Sphinx3", and "Apache" and showed that it has negligible overhead. Moreover, we uncovered two unreported asymmetric data races.

CHAPTER 5

IMPLEMENTING A FLEXIBLE HARDWARE SIGNATURE MODULE

Signatures are a hardware resource that can keep an unbounded number of addresses in a bounded space (see Section 1.5) and they can be used for many hardware tools related with parallel computer architecture, to optimize their resources and to enhance their performance. Examples of these tools are TM systems, data race detectors, deterministic replay or code analysis and optimization.

A drawback of hardware signatures is the lack of flexibility. If signatures are designed for a specific purpose or application (with a specific size, number of hashes, etc.), probably doesn't fit well for other different purposes. The contribution of the thesis presented in this chapter contributes to make signatures more flexible, proposing a new hardware signature module specifically designed to work in a concurrent environment, that we call *FlexSig* [122]. The aim is to make the best use of the hardware resources available, by hosting a large number of signatures in a limited and reduced amount of hardware resources, and to achieve a low false positive rate. This chapter presents a basic *FlexSig* module with symmetric allocation algorithms, whereas in Chapter 6 we present a more advanced *FlexSig* with asymmetric allocation algorithms and a higher performance parallel hardware implementation.

The chapter is organized as follow: Section 5.1 describes our flexible signature module, Section 5.2 depicts a high-level implementation of the module, Section 5.3 evaluates the proposal, Section 5.4 discusses related work and Section 5.5 concludes the chapter.

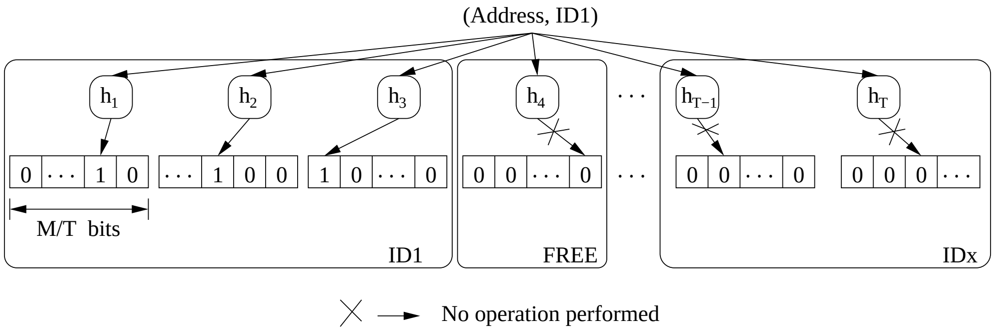


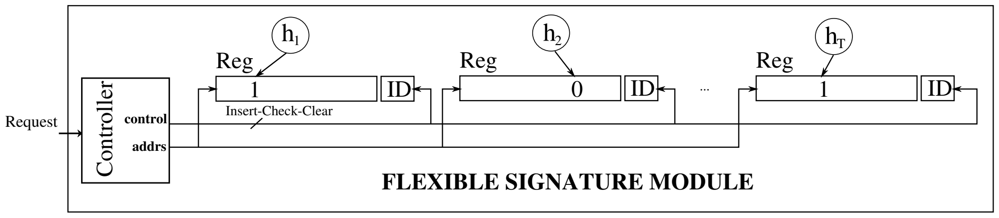
Figure 5.1: Block diagram of *FlexSig*. Each allocated signature in *FlexSig* has a variable number of hash functions k between 1 and T, depending on the number of signatures allocated concurrently. Each signature is identified with an ID. The total register space assigned to each signature is $m = k * M/T$ bits.

5.1 *FlexSig*: Implementing Flexible Hardware Signatures

FlexSig is based on parallel Bloom filters (see Section 1.5), but introducing mechanisms to use all the signature resources as much as possible and with a large flexibility to adapt to different signature demands, allowing a better efficiency and reconfigurability.

Figure 5.1 shows the block diagram of *FlexSig*. It is composed of T Bloom filters, each one composed of a M/T–bit register (M is the total size of *FlexSig*) and a hash function, that can host between 1 and T signatures, each signature being composed of one or more Bloom filters. Each Bloom Filter has an identifier (ID) of the signature to which it belongs. The number of Bloom filters assigned to each signature depends on the number of signatures allocated. Moreover, the resources assigned to a given signature may change dynamically. The modules $h_1, h_2, ...h_T$ are independent H_3 hash functions, each operating on one register. The registers in *FlexSig* are usually relatively small (for instance 64 bits), because a signature is composed of several of them. Every time *FlexSig* receives a request to insert a new address in one of its signatures, each hash function assigned to the signature sets one bit in its register. On the other hand, to check if an address is already stored in the signature, all the bits read by the corresponding hash functions should be 1. Deallocation requests clear all the IDs and registers assigned to the signature.

Each time a new signature allocation request arrives, *FlexSig* assigns k Bloom filters, $k \leq T$, to the new signature. Then, the k Bloom filters operate as a parallel Bloom filter inside *FlexSig*. The number of Bloom filters assigned depends on the current resource avail-



h_x : hash functions (there is T hash functions).
Reg: one register per hash (T registers).
Bloom Filter: we call Bloom Filter to the set composed by one register and one hash.
ID: identifier of the owner. It is composed by thID that identifies the thread, and by sID that identifies different signatures with the thread, and by sID that identifies different signatures with the same thID.
Controller: implements the FlexSig logic.
T: number of Bloom Filters.

Figure 5.2: *FlexSig* module architecture.

ability, that is, on the already allocated Bloom filters to previous signatures. If the hardware resources in *FlexSig* are fully used by previous signatures, *FlexSig* has to free several Bloom filters, already assigned, to allocate the new signature. This means that *FlexSig* has to reduce dynamically the size of any signature by releasing Bloom filters. In this case, the false positive rate may increase, but false negatives are never produced.

Figure 5.2 shows the *FlexSig* module architecture. As said before, there are T Bloom filters, each one composed of a register and a hash function. Attached to each register there is a thread identifier (thID) and a signature identifier (sID) (a thread might have more than one signature allocated), used to identify the registers assigned to a given signature. Therefore, being *num_threads* the maximum number of threads managed by the filter, and being *max_sigs_per_thread* the maximum number of signatures that a thread can allocate, thID and sID are $\log_2(num\_threads)$–bit and $\log_2(max\_sigs\_per\_thread)$–bit wide, respectively. Both thID and sID form the signature owner identifier (ID).

The maximum number of Bloom filters per signature is T (when only one signature is allocated), and the minimum size is $T/\#max\_csigs$ (the Bloom filters are distributed equally among signatures), being *#max_csigs* the maximum number of concurrent signatures in the module.

The controller implements the allocation algorithm and the rest of functions needed for the correct operation of *FlexSig*. The complexity and efficiency of the controller is determined by the algorithm to allocate signature registers.

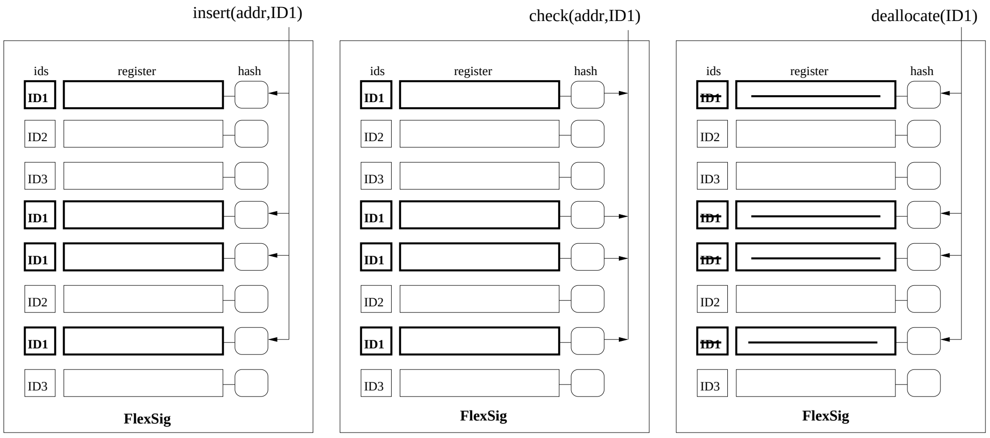


Figure 5.3: Insertion, check and deallocation request in *FlexSig*.

Figure 5.3 shows how to perform the insertion, check and deallocation requests. The insertion request must include the ID of the signature, so that the address is only inserted in the registers that matches this ID. The check operation is very similar to the insertion operation, but it is read-only. The deallocation consists of clearing the registers and IDs.

5.1.1 Allocation Algorithm

The allocation algorithm is required to make room for a new signature. This algorithm may be very complicated, for example, by defining priorities to assign more or less Bloom filters depending on the requirements of the allocated signature. In this chapter we show a simple allocation algorithm with no priorities. In Chapter 6 we will extend *FlexSig* with priorities implementing several asymmetric allocation algorithms.

Figure 5.4 shows a very simple graphical example of the allocation algorithm. At the beginning, the module is empty, and to allocate the new signature ID1, the controller just assigns all the resources to it. Next, to allocate ID2, the controller calculates the number of necessary resources for the new signature (8 Bloom filters) and it frees them from ID1. Finally, to allocate resources for ID3, the controller calculates the number of necessary resources for the new signature (5 Bloom filters) and it frees them from ID1 and ID2.

To illustrate the advantages of *FlexSig*, the same example but in a system with conventional signatures (implemented with Bloom filters) is shown in Figure 5.5: a fixed number of

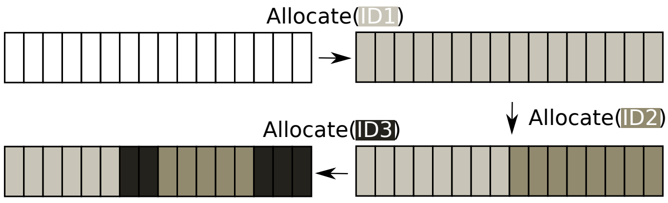


Figure 5.4: Example of the symmetric allocation algorithm of *FlexSig*.

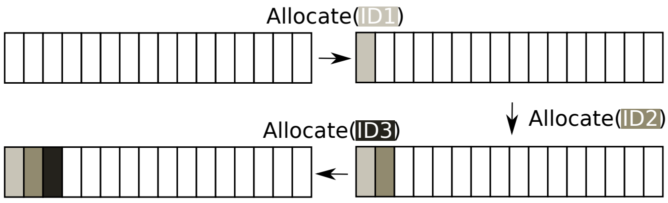


Figure 5.5: Example of conventional signatures used in a system with a maximum of 16 simultaneous signature requesters.

Bloom filters are assigned in each allocation request (only one Bloom filter in this case, to allow a maximum of 16 concurrent allocated signatures, the same as in the *FlexSig* example of Figure 5.4). This example illustrates the advantage of *FlexSig* with respect to conventional signatures, that make an inefficient use of resources because only for maximum concurrency (when the number of concurrent allocated signatures is 16 in this example) all the resources are used.

Therefore, by generalizing the example of Figure 5.4, the controller of *FlexSig* should determine the average number of Bloom filters per signature taking into account the incoming request, that is T/n_sig, with T being the number of Bloom filters and n_sig the number of signatures. Since T may not be a multiple of n_sig, in general it is not possible to assign exactly the same number of Bloom filters to each signature (for example the Allocate(ID3) in Figure 5.4). Moreover, there might be signatures in *FlexSig* with fewer than T/n_sig Bloom filters, due to previous allocations and deallocations and the fact that signatures can only reduce their size, but are not allowed to grow.

One implementation that seeks to allocate resources as evenly as possible, would use the following policy to take into account these factors. The *FlexSig* controller computes the integer value of the average number of Bloom filters per signature

$$nb = \left\lfloor \frac{T}{n\_sig} \right\rfloor \tag{5.1}$$

and determines the value of T' and n_sig', with T' being the total number of Bloom filters in *FlexSig* that are allocated to n_sig' signatures with more resources than nb. Then, it assigns

$$nb' = \left\lfloor \frac{T'}{n\_sig'} \right\rfloor \tag{5.2}$$

Bloom filters to $n\_sig' - (T' \mod n\_sig')$ signatures, and it assigns $nb' + 1$ Bloom filters to $(T' \mod n\_sig')$ signatures, where $\mod$ is the modulo operator. The implementation of this policy in the *FlexSig* controller is complex and may involve several cycles for the allocation. As an alternative, we propose a simple method described below.

To perform the description of the allocation algorithm, we define a new parameter nb_free, that is the number of free Bloom filters in *FlexSig*.

Three different situations are possible when a thread tries to allocate space for a new signature in *FlexSig*: (1) *FlexSig* is empty, (2) *FlexSig* is full, or (3) *FlexSig* is partially full.

1. ***FlexSig* is empty** ($n\_sig = 1$, $nb\_free = T$). All the resources of *FlexSig* are assigned to the new signature. This is one of the basic principles of *FlexSig*: if there are free resources, take as much as possible.

2. ***FlexSig* is full** ($n\_sig > 1$, $nb\_free = 0$). The controller must free space in *FlexSig* when it is necessary to allocate a new signature. Then, the other signatures in *FlexSig* are made smaller by reducing the number of Bloom filters per signature. The release of one or several Bloom filters assigned to a given signature may increase the false positive rate but false negatives are never produced, because all the hash functions are independent and all the registers in a signature have the information corresponding to every address inserted. The number of Bloom filters to free is given by nb (Equation (5.1)). The free filters are assigned to the new signature. Therefore, the filters are redistributed among all the signatures including the new one.

3. ***FlexSig* is partially full**. The controller must decide whether the free resources are enough to allocate a new signature or if additional resources are needed. In the latter case, some filters should be freed and assigned to the new signature. If $nb\_free < nb$, the controller frees $(nb - nb\_free)$ Bloom filters as explained for the case when *FlexSig*

is full. On the other hand if $nb\_free \geq nb$ all the available Bloom filters are assigned to the new signature.

Notice that, despite *FlexSig* tries to distribute the resources equally among all the signatures, there can be situations where signatures have assigned a number of Bloom filters different among each other. This is due the fact that the signatures can not grow in size to balance the resources in the allocation algorithm.

5.1.2 Influence of the Bloom Filters Release on the False Positive Rate

As said before, when a new signature is needed and there is not room to host it, the controller must free some Bloom filters assigned to other signatures and assign them to the new signature. But, how does this affect the false positive rate?

The lower bound probability of a false positive is $P_{FP} = (1-(1-k/m)^n)^k$ (see Section 1.5), being m the number of bits of the signature, k the number of registers or hash functions, and n the number of elements inserted in the signature. In *FlexSig*, the relation m/k is constant. When the number of Bloom filters assigned to a signature is reduced, m and k are reduced in the same proportion.

Let us illustrate this influence with an example. Figure 5.6 shows the variation of the false positive rate when the resources allocated to a given signature are reduced: as an instance, assume that initially the signature is composed of $k = 16$ filters with a total register size of $m = 2048$ and then it is reduced to $k = 8$ filters with $m = 1024$. If the number of addresses inserted in the signature, n, is low, the reduction in signature size has no practical influence on the false positive rate. However, if n is large the false positive rate increase significantly (for instance, in a signature with $k = 4$, $m = 512$, for a value of $n = 250$ the false positive rate is 54,5%).

5.1.3 Software Interface

To provide a basic software interface to *FlexSig*, the following instruction set extensions should be added:

- **Allocate(ID)**: allocate the signature with identifier ID. The size of the allocated signature is defined by the controller.

- **Deallocate(ID)**: deallocate the signature with identifier ID.

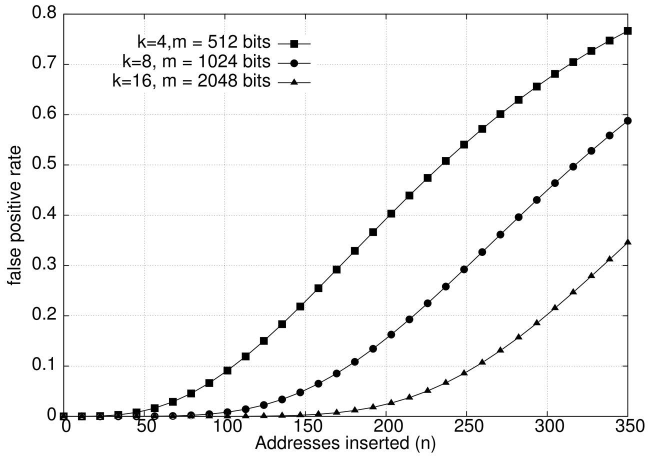


Figure 5.6: False positive rate in *FlexSig* for different signature sizes.

- **Insert(addr, ID)**: insert the address *addr* in the signature with identifier ID.
- **Check(addr, ID)**: check if address *addr* was inserted previously in the signature with identifier ID.

This instruction set extension allows the proper interface to the basic operations of the module. Moreover, the instruction set can be extended with new instructions for specific purposes. For instance, to support TM systems, it can be extended with functionalities to forward signatures to cores, etc. Alternatively, this actions can be implemented as writes to memory-mapped registers, which allows not to modify the instruction set architecture.

5.1.4 Register Grouping

In the case of just one or few signatures allocated in *FlexSig*, the number of Bloom filter elements per signature is high, i.e. equivalent to having a high value of k in conventional parallel Bloom filters. However, as we learned from Figure 1.10(b) in Chapter 1, the optimum value of k, in terms of the false positive rate, is low (between 3 and 8 for the parameters used in Figure 1.10(b)). To reduce the false positive rate in these cases, *FlexSig* can group several Bloom filters so that only one is used at a time in operations that involve hash functions (insert and check). A simple implementation consists of selecting the Bloom filter of the group based

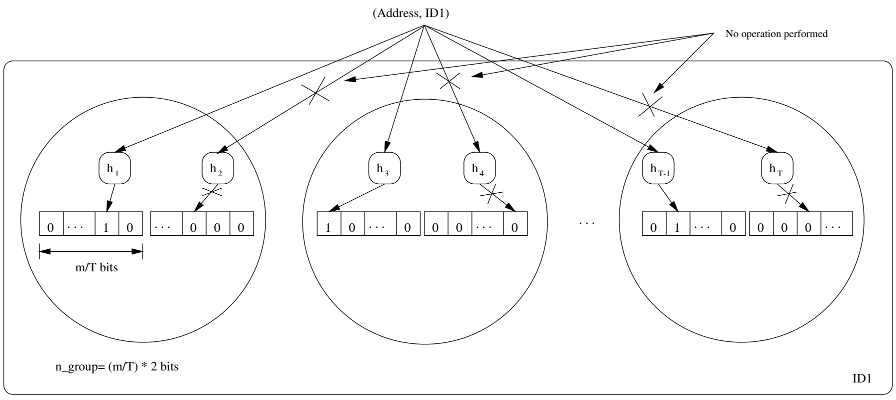


Figure 5.7: Example of register grouping, for the case of only one signature allocated and grouping of two elements.

on the value of a few least significant bits of the address involved in the operation. Figure 5.7 illustrates the grouping scheme for groups of two elements, being each element m/T bits wide, and each group $n\_group = (m/T) * 2$ bits wide.

Grouping can be implemented in a static or dynamic way. For a static implementation, the grouping size is chosen before the first allocation, and only can change when the *FlexSig* is totally empty. This forces all signatures to have the same grouping size, simplifying the implementation. If it is implemented dynamically, the grouping size is chosen for each signature when it is allocated, depending on the number of Bloom filters assigned to it. This complicates the logic and can cause other problems with many corner cases. Because of that, we chose to implement static grouping in our evaluation.

The maximum level of grouping is a design decision for *FlexSig*. The specific grouping for each application can be established through the software interface. For the case of our benchmarks (see Section 5.3.3) with up to 16 threads, we determined that a maximum grouping of two elements is enough to achieve good results. Moreover, this grouping is activated only for applications configured to run with two and four threads.

5.1.5 *FlexSig* Overflow and Fault Tolerance

FlexSig allows to host several signatures at the same time. However, a situation of overflow may be produced in exceptional (low probability) cases when a new allocation request arrives, and the controller can not free any Bloom filter because *FlexSig* hits the maximum number of signatures allowed (when the software tries to allocate more signatures than the total number of Bloom Filters available). In this case, the situation is managed by software, signaling an exception that jumps to a pre-registered software handler.

In the case that an application allocates signatures, but fails to deallocate them (due to a software bug or fault), *FlexSig* will have fewer resources to allocate new signatures for the remaining running application time (similar to the memory leak problem). This case is very hard to manage in hardware, and therefore it should be handled by software. As an instance, a straightforward scheme for TM systems is to clear *FlexSig* when serial code is executing or when no transactions are running in the system.

FlexSig has nice fault tolerant properties regarding the storage of the signatures due to its flexibility. If one register fails (permanent or soft error detected with standard fault detection techniques), such register is marked as invalid if the error is permanent (not used any more) or freed if it is a soft error (it can be used in new signature allocations). Moreover, regarding permanent faults, only stuck-at-zero faults would lead to the invalidation of a register (stuck-at-one faults only increase the false positive rate). No special operations are needed for managing this situation, as the only implication of loosing one Bloom filter is to increase the false positive rate. Of course, an exception is raised if the failing Bloom filter is the only one assigned to the signature.

5.2 Implementation Issues

The *FlexSig* system can be placed in each core, or as a centralized resource attached to the chip interconnection network, or it can be distributed among directories in a system with a directory based cache coherence protocol (for instance, each *FlexSig* module serving a cluster of cores). The flexibility of *FlexSig* is achieved at the cost of extra logic compared with conventional parallel Bloom filters. In this sense, a key element is the *FlexSig* controller, which should support the functionality of *FlexSig* with a simple architecture to reduce power an area overhead.

The controller needs one queue for the incoming requests, because the requests are served sequentially. There are four types of requests, *Allocate*, *Deallocate*, *Check* and *Insert*, each with an ID that the controller uses to take action on the corresponding registers. To implement efficiently the straightforward allocation algorithm described in Section 5.1.1, some extra registers are needed in the controller to take fast decisions for the allocation operation. There are

$T\ \log_2(T)$-bit counters in the controller to count the number of registers of *FlexSig* allocated by the corresponding ID. There is also a counter that keeps track of empty records. Using this stored information a finite-state machine performs the allocation operations.

For implementations using a centralized *FlexSig* for all cores, the module might be a bottleneck. After analyzing the concurrency of the possible arriving requests, we determined that the controller can serve some of them in parallel. Specifically:

- **Insert** : it can be executed in parallel with other *inserts*, *checks* or *deallocates* with different IDs.
- **Check** : it can be executed in parallel with other *inserts* and *deallocates* with different IDs and with other *checks* with any ID.
- **Allocate** : it can not be executed in parallel with other requests.
- **Deallocate** : it can be executed in parallel with *deallocates*, *inserts* and *checks* with different ID.

Taking into account these rules, the controller can be parallelized in several ways. Figure 5.8 shows a simple parallel controller proposal. This controller may perform up to *P* operations in parallel. Basically it is an in-order issue superscalar engine. The incoming requests are placed in an input queue. The issue logic determines up to *P* requests to be issued in parallel, following the rules listed above. We have one finite-state machine (and the corresponding counters) to execute the *allocate* requests, and *P* very simple circuits to process *inserts*, *checks* or *deallocates*.

Most of the time, the finite-state machine has the calculations ready when a new *allocate* request arrives, because it recalculates these parameters immediately after the previous allocate or deallocate request. Only when two consecutive allocate requests arrive, the finite-state machine has no time to recalculate the parameters before the second request, incurring in some additional delay.

5.3 Evaluation in a TM System

The aim of the evaluation is to show the effectiveness of the *FlexSig* system with respect to conventional parallel Bloom filters. In this work we concentrate on TM applications, since for many TM implementations signatures are a key element. Some TM systems use signatures

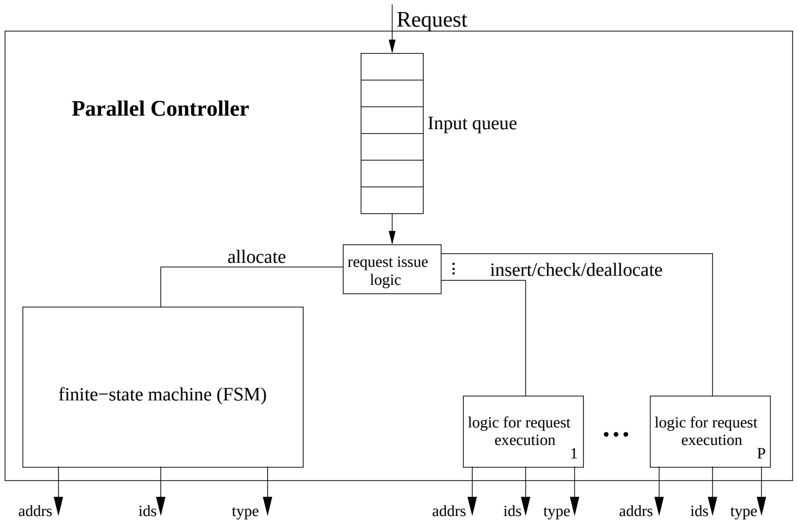


Figure 5.8: Parallel controller implementation.

to detect conflicts among transactions. Each transaction inserts in signatures its read and write addresses to maintain a summary of its read/write set. Conflicts with other transactions reads/writes are detected through the check operation. Since our purpose is to evaluate only signatures, our figure of merit is the false positive rate of the signature system. Higher false positive rates degrade performance, because for each false positive, the TM system has to do an unnecessary abort (rollback to the initial state and restart the transaction).

For simplifying the allocation algorithm, we use unified signatures (see Section 5.3.1) to evaluate *FlexSig*, so we only need one signature per transaction for the read and write set, which allows us to implement the simple allocation algorithm described in Section 5.1.1.

5.3.1 Unified signatures: Simplifying *FlexSig* Implementation in TM

TM uses two signatures per transaction, one for the read set and another one for the write set. Usually the read set is larger than the write set, and therefore, in order to use efficiently the resources, the signature of the read set should be larger than the signature of the write set. However, having signatures of different sizes for the write set and the read set introduces additional difficulties in the allocation algorithm and makes its implementation more complex. Unified signatures [36] propose to use only one signature for both the read set and the write

Bench.	input
Genome	-g128
Intruder	-a10 -l16 -n4096 -s1
Kmeans-high	-m15 -n15 -t0.05 -i random-n2048-d16-c16.txt
Kmeans-low	-m40 -n40 -t0.05 -i random-n2048-d16-c16.txt
Labyrinth	-i random-x256-y256-z3-n256.txt
Ssca	-s14 -i1.0 -u1.0 -l9 -p9
Vacation-high	-n4 -q60 -u90 -r1048576 -t4096
Vacation-low	-n2 -q90 -u98 -r1048576 -t4096
Yada	-a10 -i ttimeu10000.2
Streamcluster	10 20 32 4096 4096 1000
Canneal	2000 2000 10.nets

Table 5.1: Benchmark Inputs.

set. This approach may generate read-read conflicts, however, these conflicts rarely lead to a performance lost [148][36]. Using unified signatures each thread only needs to allocate one signature per transaction, and the complexity of the controller is reduced. This is the approach we have used for evaluating *FlexSig*.

5.3.2 Experimental Setup

To evaluate the *FlexSig* scheme we use a TM system with signatures used to track data accesses in transactions. Our aim is not to implement a fully functional TM system, but to work out a challenging scenario for *FlexSig*, and compare it with conventional parallel Bloom filters in the same situation. For the TM system we use the software implementation RSTM [159]. RSTM is a software TM system that allows many different configurations. In our evaluation we use a lazy acquisition and lazy versioning with extendable timestamps [143] to configure RSTM. We use PIN [81] to track all transactions and memory accesses of RSTM and to emulate the hardware signatures. This conforms the simulation of a Hybrid Transactional Memory system.

We run several benchmarks over the RSTM system. Specifically, we use all the STAMP Benchmarks [25], two PARSEC Benchmarks [16] and nine micro benchmarks (included in the RSTM distribution). Table 5.1 shows the inputs of the benchmarks. The benchmarks not included in the table run with the default input. For this evaluation, we classify the bench-

Benchmark	#Tx	TxTime	RS	WS
Intruder	101780	32%	19	2
Vacation-high	4096	94%	384	7
Vacation-low	4096	94%	283	5
Yada	14316	68%	59	17
LinkedList	175	52%	141	0.3
DList	152	55%	138	0.6
PrivList	94	81%	256	1

Table 5.2: Benchmark set A, characterization with 16 threads.

marks in two categories. One group is composed by benchmarks with a high false positive rate (Benchmark set A), and the other with a modest false positive rate (Benchmark set B). The purpose of this is to run each group of benchmarks with a different signature configuration to show the advantages of *FlexSig* for workloads with different characteristics.

Tables 5.2 and 5.3 show the characterization of the benchmarks. The parameter $\#Tx$ is the number of transactions of the benchmark, $TxTime$ is the percentage of time spent on transactions, and *RS* and WS are the average number of reads and writes per transaction. The time spent in transactions is, in general, very significant. This parameter is affected by the instrumentation tool, because only transactions are instrumented. This scenario is a pessimistic approximation, since in a real system the time spent inside the transactions should be less, and therefore, it should be less likely that those transactions demand signatures at the same time in the *FlexSig* system. Therefore, the results should be better than in the simulated case.

5.3.3 Configuration

For the evaluation we use the configurations shown in Table 5.4. The hardware configuration for parallel Bloom filters (k and m are the parameters in Figure 1.9) was chosen specifically to manage up to 16 threads (that is, the conventional signature system has 16 parallel Bloom filters of fixed size). We run experiments with 2, 4, 8 and 16 threads. Two configurations are used for *FlexSig*: configuration *conf1* uses the same resources as their equivalent parallel Bloom filter, and *conf2* uses half of the resources. For the benchmarks belonging to the set A, the registers are of 512 bits for the unified parallel Bloom filter (a total of 8192 bits for a 16 thread system); for *FlexSig* we have 32 registers of 128 bits for *conf2*, and 64 registers of 128

Benchmark	#Tx	TxTime	RS	WS
Bayes	644	46%	8	2
Genome	353994	76%	26	0
Kmeans-high	8238	43%	13	13
Kmeans-low	8557	70%	13	13
Labyrinth	544	54%	84	80
Ssca	93731	49%	1	2
Streamcluster	592	17%	1	0
Canneal	4000	44%	2	1
Counter	759	23%	1	1
HashTable	2772	47%	2	0.3
RBTree	16385	68%	18	2
RBTreeLarge	134	61%	27	3
LFUCache	62	61%	7	2
RandomGraph	53	59%	506	2

Table 5.3: Benchmark set B, characterization with 16 threads.

Signature	Description
Unified Parallel Bloom (set B)	16 registers with k=4 and m=32 bits (512 bits total)
Unified Parallel Bloom (set A)	16 registers with k=4 and m=512 bits (8192 bits total)
Unified *FlexSig* conf2 (set B)	32 registers of 8 bits (256 bits total)
Unified *FlexSig* conf1 (set B)	64 registers of 8 bits (512 bits total)
Unified *FlexSig* conf2 (set A)	32 registers of 128 bits (4096 bits total)
Unified *FlexSig* conf1 (set A)	64 registers of 128 bits (8192 bits total)

Table 5.4: Configuration used with unified signatures.

bits for *conf1*. Similarly, for the benchmarks belonging to the set B, the size of the registers for the unified parallel Bloom filter is 32 bits and the corresponding *FlexSig* configurations *conf1* and *conf2* are described in Table 5.4. To group registers (see Section 5.1.4), we choose groups of one register for 8 and 16 threads, and groups of two registers for executions with 2 and 4 threads. This decision was taken to have an efficient configuration (see Figure 1.10(b)).

Benchmark	2 threads			4 threads			8 threads			16 threads		
	Bloom	conf2	conf1	Bloom	conf2	conf1	Bloom	conf2	conf1	Bloom	conf2	conf1
Intruder	1.5	0.0	0.0	2.0	0.2	0.0	2.4	2.4	0.2	2.9	12.0	2.7
Vacation-high	38.1	3.7	0.5	37.9	13.0	3.7	38.1	38.0	24.2	38.3	52.9	38.2
Vacation-low	25.3	0.8	0.0	25.3	6.0	0.8	25.4	25.3	11.1	25.2	42.5	25.1
Yada	18.8	0.6	0.0	19.7	4.7	0.7	20.4	20.4	8.9	20.1	34.4	20.1
LinkedList	4.6	0.1	0.0	2.6	0.4	0.0	1.5	1.5	0.2	0.7	4.7	0.7
DList	4.0	0.1	0.0	2.9	0.5	0.0	2.0	2.0	0.2	0.7	2.9	0.7
PrivList	6.2	0.5	0.1	8.1	3.8	1.5	3.5	3.5	2.1	4.2	7.1	4.2

Table 5.5: Benchmark Set A. False positives comparison (in %) for Unified Signatures.

Benchmark	2 threads			4 threads			8 threads			16 threads		
	Bloom	conf2	conf1	Bloom	conf2	conf1	Bloom	conf2	conf1	Bloom	conf2	conf1
Bayes	43.2	7.1	3.0	33.1	13.0	6.5	32.4	31.7	23.5	24.9	34.1	24.4
Genome	41.8	4.8	0.7	40.7	13.6	4.6	43.3	42.0	29.2	9.7	54.4	9.4
Kmeans-low	45.1	11.2	1.0	41.2	17.9	10.6	40.0	40.0	36.4	36.7	69.8	36.6
Kmeans-high	40.6	7.6	0.7	37.6	16.4	8.3	36.6	36.6	33.8	37.5	66.1	37.4
Labyrinth	19.0	17.9	14.7	16.7	15.8	15.6	41.9	41.9	41.6	72.2	77.3	72.2
Ssca	1.1	0.0	0.0	1.1	0.1	0.0	1.2	1.1	0.0	1.2	7.7	1.1
Streamcluster	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Canneal	3.3	0.0	0.0	3.2	0.0	0.0	3.5	3.5	0.5	3.3	9.6	3.1
Counter	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
HashTable	2.5	0.0	0.0	2.0	0.1	0.0	1.9	1.9	0.2	1.9	8.8	1.7
RBTree	49.7	16.7	8.3	53.6	30.8	19.7	46.8	46.8	40.3	44.0	50.1	44.0
RBTreeLarge	48.8	16.5	8.3	47.1	25.1	16.3	44.5	44.5	35.8	37.4	46.4	37.3
LFUCache	26.2	8.8	4.7	28.3	12.8	10.5	29.7	29.7	26.2	27.6	31.6	27.6
RandomGraph	17.9	13.4	11.8	17.0	12.7	10.7	16.6	16.6	14.4	16.9	20.3	16.9

Table 5.6: Benchmark Set B. False positives comparison (in %) for Unified Signatures.

5.3.4 Results

Tables 5.5 and 5.6 show the false positive rate of *FlexSig* with configurations *conf1* and *conf2* compared with the results obtained with parallel Bloom filters (see Table 5.4), for the case of 2, 4, 8 and 16 threads. A white cell (in *conf1* and *conf2* columns) means that the false positive rate is roughly the same as the one obtained with the parallel Bloom filter, a gray cell means that the false positive rate of *FlexSig* is better (lower), and a dark gray means that the false positive rate of *FlexSig* is worse (higher).

First, we comment the results with *conf1* for both benchmark sets A and B. As Tables 5.5 and 5.6 show, the *FlexSig-conf1* outperforms parallel Bloom filters in almost all the cases. For 2, 4 and 8 threads, the improvement is very high; for instance, for the case of "Vacation-high" running with 2 threads, the false positive rate is reduced from 38,1% to 0,5%. As the number of threads increases, the advantage of *FlexSig* decreases. However, even in the worst case

(16 threads), *FlexSig* improves with respect to conventional Bloom filters in many cases, and never performs worse. The results are better as fewer threads are running, because *FlexSig* tries to use all the resources, while the parallel Bloom filter implementation has fixed size for each signature independently of the number of threads.

The results of *FlexSig-conf1* with 16 threads are very similar to the implementation with parallel Bloom filters since for this case all the signatures are used. *FlexSig* achieves better results because not all the threads allocate signatures at the same time, and it can use the free resources also in this case. The results are only slightly better because the benchmarks are highly concurrent (in part due to the instrumentation performed by PIN).

FlexSig-conf2 uses half of the resources of the parallel Bloom filter implementation. Even with this configuration, *FlexSig* clearly outperforms the parallel Bloom filter implementation for 2 and 4 threads. For instance, for *"Vacation-high"* the false positive rate with two threads is reduced from 38,1% to 3,7%. For the case of 8 threads, the results are similar in both implementations, but *FlexSig* outperforms the parallel Bloom filter implementation in many cases, and at least matches it. For 16 threads, *FlexSig* has worse performance, but it has the flexibility to manage the 16 threads with half the resources.

It is of interest to show the reduction of the number of false positives in absolute terms, since each false positive may lead to an unnecessary abort in a TM system. Figure 5.9 shows the percentage of decrease of the number of false positives in *FlexSig-conf1* compared with conventional signatures for all benchmarks, which is specially high for 2,4 and 8 threads. Figure 5.10 shows the percentage of decrease of the absolute number of false positives in *FlexSig-conf2* (with half of the resources of conventional signatures) compared with conventional signatures for all benchmarks, with very good results, except for 16 threads (the negative values represents – changing the sign to positive – the percentage of decrease of the absolute number of false positives in conventional signatures compared with *FlexSig-conf2*). For "StreamCluster" and "Counter" benchmarks, the number of false positives does not decrease in Figure 5.9 nor Figure 5.10, because in both simulations the number of false positives is zero for *FlexSig* and conventional signatures.

Notice that the advantage of *FlexSig* when compared to conventional signatures is reduced as the number of threads increases. As we show in Section 6.3.5 (for *FlexSig* supporting priorities), this is not due to any scalability issue. The reason is that *FlexSig* always take advantage of all the resources of the module for any number of threads, and therefore increasing the number of threads reduces the average size of signatures for each thread.

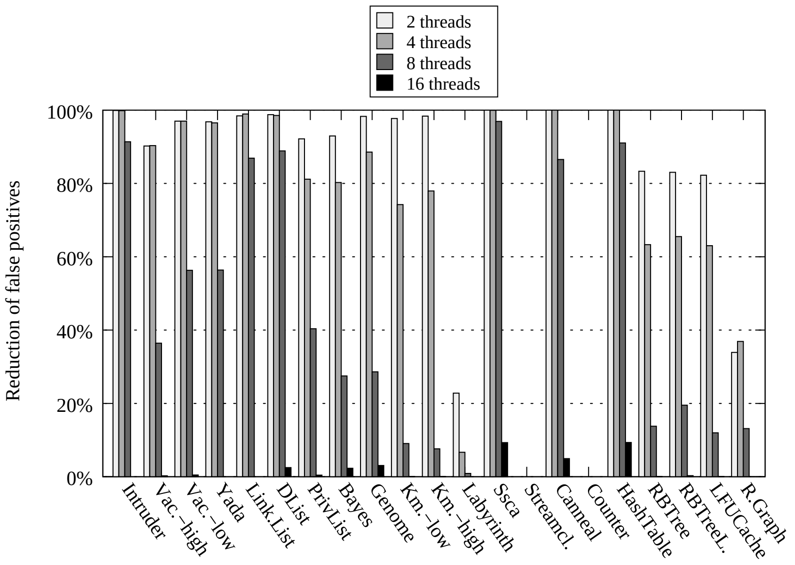


Figure 5.9: Percentage of decrease of the absolute number of false positives in *FlexSig-conf1* compared with conventional signatures for all benchmarks.

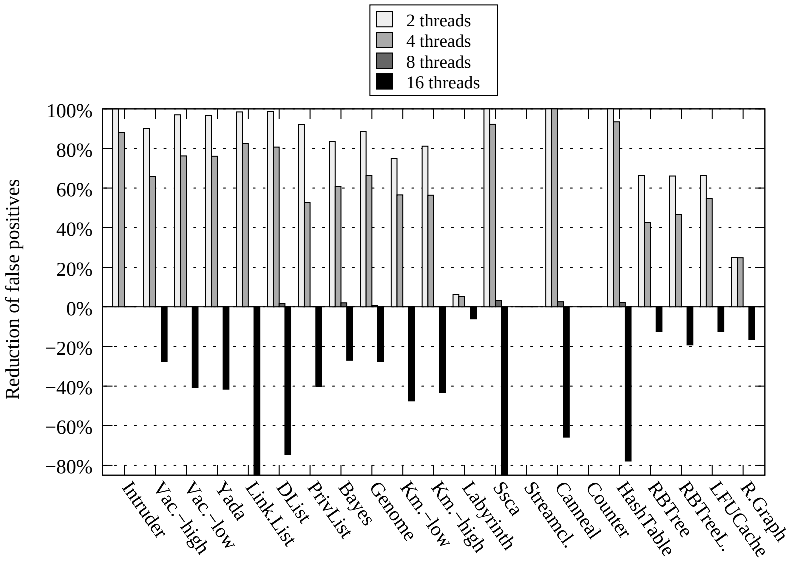


Figure 5.10: Percentage of decrease of the absolute number of false positives in *FlexSig-conf2* compared with conventional signatures for all benchmarks.

It is of interest to have an estimation of the average signature size that is used per transaction in *FlexSig*. The average signature size is the weighted average in time of the signature size, and is given by

$$ave\_sig\_size = \frac{\sum_{e=1}^{num\_changes\_size} sig\_size_e * time\_interval_e}{\sum_{e=1}^{num\_changes\_size} time\_interval_e} \tag{5.3}$$

where *num_changes_size* is the number of times a signature changes its size (number of registers) before deallocation, and *time_interval* is the number of time units that a signature has a size *sig_size*.

Figure 5.11 shows the improvement in the average signature size of *FlexSig-conf1* running 16 threads compared with conventional signatures. The configuration of the signatures is shown in Table 5.4. This figure tries to show the advantage of *FlexSig* over conventional signatures with the same resources than *FlexSig* even in the situation where conventional signatures can use all their resources (in this case, with 16 running threads). Moreover, in a situation with less threads running, the advantage of *FlexSig* is much bigger. The improvement shown in the figure is achieved because not all the threads use signatures simultaneously, and therefore, threads can take resources that others are not using. The average signature size for *FlexSig* depends basically on the concurrent nature of the benchmark (less concurrent threads lead to a better performance of *FlexSig*). The best result in terms of *FlexSig* average signature size improvement is for "Streamcluster", with a more than 50% improvement, due to the low transaction concurrency in this benchmark. In this case, the improvement of the signature size doesn't imply a significant reduction of the false positive rate because this is already very low in absolute terms.

As a conclusion, the *FlexSig* system improves the false positive rate when compared with conventional parallel Bloom filters. In a system configured to run 16 threads, our signature system clearly outperforms the parallel Bloom filter implementation when the number of threads is lower than 16 (for the *conf1* with the same resources), due to the flexibility of *FlexSig* to assign the physical resources depending on the demand (number of concurrent threads). General purpose multicore and multiprocessors are able to run a large number of concurrent threads, but many applications use only a few threads. *FlexSig* is flexible enough to provide these applications all the available signature resources to achieve better performance. We used very hard conditions in our evaluation to demonstrate that *FlexSig* can perform well even in an unfavourable scenario. The benchmarks used are highly concurrent, which means that many transactions use signatures at the same time. Moreover, because of the instru-

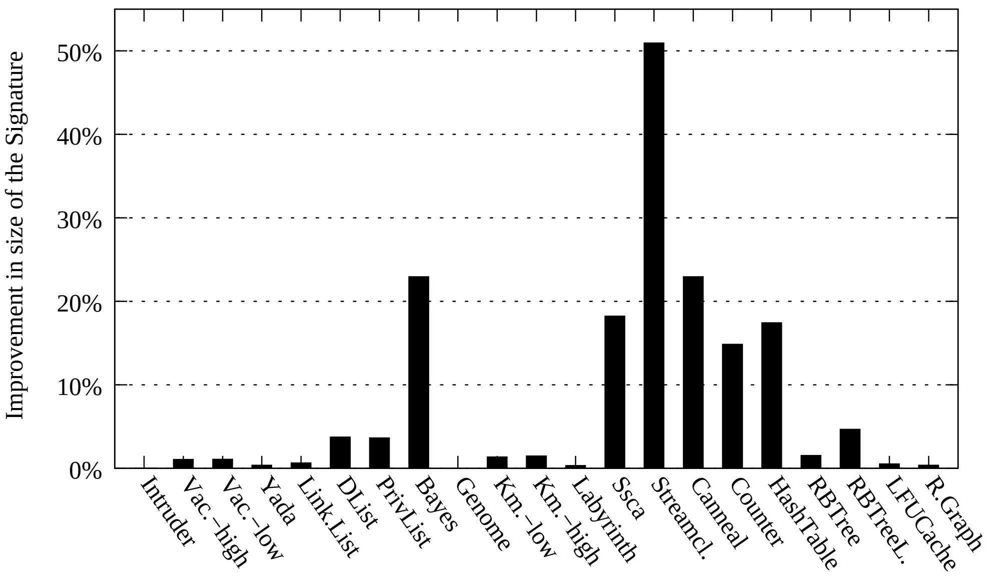


Figure 5.11: Increment of the average signature size in *FlexSig-conf1* with 16 threads compared to regular signatures for Benchmark Set A and Set B.

mentation tool, the benchmarks spend more time inside transactions, increasing transactional concurrency.

5.4 Related Work

Most of the papers dealing with signatures are focused on improving performance, reducing chip area or reducing the false positive rate [147][134][182][154][184]. However none of these papers focus on flexibility and scalability. The Scalable Bloom Filters (SBF) proposed by [4] tries to make an approximation of scalable signatures. They use one signature, and when a fill ratio is reached, another signature is used. SBF was proposed to avoid the problem of oversize signatures due to the fact that the size of the signature must be defined previously based on the number of elements to be stored and the desired upper bound of the false positive rate. The SBF method can reduce the specific size of the signature used. However, it may use several signatures depending on the number of elements to be stored, and therefore, in reference to our context, the system has to be oversized anyway (with regard to the number of signatures). *FlexSig* does not fully avoid the problem of oversized signatures, but it is more flexible and efficient in the sense that it uses as many hardware resources as possible, having a significant effect on the false positive rate for a TM system.

In recent publications we find TM systems that fit very well for using *FlexSig*. In [104] proposes a STM system with a centralized conflict detection mechanism (based on software signatures) placed in one core. One way to improve this scheme would be to use *FlexSig* instead of their software signatures. This would improve performance maintaining the flexibility of the software signatures. Another example is the scheme proposed by [28], that describes a new centralized hardware outside the processor chip to accelerate STM systems. This special hardware includes signatures. They also propose two algorithms for conflict detection, one using two signatures per transaction and other using three signatures. *FlexSig* would allow to implement both with the same hardware and also performance would be improved.

The idea behind *FlexSig* is similar to the recent trend of incorporating a shared last level cache in multicore systems. The cache size used by each core varies dynamically depending on the application. This leads to a more flexible system than having a fixed size slice of the last level cache assigned to each core. *FlexSig* follows this trend for a resource that might be of interest for future multicore implementations.

5.5 Conclusions

In this chapter we propose a module for hardware signatures to improve conventional signatures in terms of flexibility, scalability and fault tolerance. The main feature of *FlexSig* is that it can host a high number of signatures for cases with applications with a high number of threads and significant contention, and for the cases for low contention or few threads, it can achieve a very low false positive rate.

We described the module and its implementation, defined a detailed algorithm to allocate signatures and evaluated *FlexSig* in the context of a TM system and compared it to an implementation with conventional parallel Bloom filters. From the evaluation performed, we show that, when the number of threads is low, *FlexSig* achieves a significant improvement because of the flexibility to use all the available resources. When the number of threads is high, the results are similar to the conventional implementation due to the highly concurrent nature of the benchmarks. However, with the same amount of resources, *FlexSig* never behave worse than conventional parallel Bloom filters.

FlexSig makes signatures more flexible to use as a general purpose hardware resource, since it is able to adapt to the concurrent demand of signatures, and decouples, to some extent, the type of benchmark from the hardware.

CHAPTER 6

ASYMMETRIC ALLOCATION IN A FLEXIBLE SIGNATURE MODULE

In Chapter 5 we propose *FlexSig*, a new shared signature module system designed to make hardware signatures more flexible. The goal is to provide a supporting building block for the different parallel multicore programming tools and applications to make an efficient use of signatures.

The algorithms explored in Chapter 5 are symmetric (they assign the same resources to all the signatures). In this chapter we explore new asymmetric allocation algorithms and their hardware implementation to reduce false positives in *FlexSig* by exploiting the asymmetry present in applications and tools. *FlexSig* with asymmetric allocation policies achieves important reductions of false positives compared with symmetric *FlexSig* and conventional signatures. We concentrate on TM as a driver application, although *FlexSig* with asymmetric allocation algorithms can be used for other applications as well. The reduction of false positives has a direct impact on the performance of the applications, that in case of TM, leads to reduce the unnecessary aborts an re-execution of transactions.

This chapter is organized as follows. In Section 6.1 we present our new asymmetric allocation algorithms. Section 6.2 discusses implementation issues leading to a highly parallel hardware implementation. Section 6.3 is devoted to the evaluation of the proposed system. Section 6.4 reviews related works and finally, Section 6.5 presents the conclusions.

6.1 Asymmetric Policies

In the *FlexSig* module described in Chapter 5, the assignment of resources follows a single policy for all allocations. In this chapter, we propose new asymmetric allocation algorithms for *FlexSig* with a set of priorities to take advantage of asymmetric characteristics in applications or among applications. With an asymmetric strategy the controller assigns signatures with a different number of Bloom filters depending on the priority. As we show in Section 6.3, the asymmetric policies lead to a very significant reduction in the number of false positives compared to the symmetric *FlexSig* implementation.

There are many opportunities to explore asymmetric allocation algorithms. In a general purpose processor, usually there are several applications running at the same time, which might require different signature sizes. Moreover, an application may require several signatures for different tasks with different demands in terms of resources. The design space of *FlexSig* considering priorities in allocations is very wide and may severely affect the complexity of the allocator. Some of the parameters to consider are: number of states for a priority class, support for several orthogonal priority classes, variable or fixed ratio of resources in a priority class, etc.

In this work, inspired by the specific characteristics of TM applications we use two orthogonal priority classes, and each one with only two possible states: high priority and low priority. In a signature-based TM system, a thread executing a transaction requests the simultaneous allocation of two signatures: one to keep the read set and the other to keep the write set. Therefore two orthogonal priority classes arise naturally: a priority class for distributing Bloom filters among transactions (for instance, large versus small transactions), and an inner priority class for distributing the Bloom filters allocated to a transaction among the read set signature and the write set signature (usually read sets are larger than write sets, so a possible policy is to assign a higher priority to the read set signature; however, this is not always the case).

As we show in Section 6.2.2 these design decisions lead to a reasonable complexity increase in the allocator compared to the case of *FlexSig* without priorities. However, this simple priority scheme allows a significant reduction in the number of false positives (see Section 6.3) for TM applications. More involved priority schemes lead to a very complex allocator.

6.1.1 Asymmetric Allocation Algorithm with Two Priority Classes

The general scheme with two orthogonal priority classes is as follows. An agent (typically a thread) issues an allocation request to *FlexSig* to allocate signatures. One of the priority classes (we call it Outer Priority Class - PCOUT) is used for determining the total number of Bloom filters assigned to the agent request. Regarding the PCOUT priority class, agents are classified as of high and low priority, and there can be any number of agents of different priorities. The other priority class (we call it Inner Priority Class - PCIN) is used to distribute the Bloom filters assigned to the agent among its internal signatures. Regarding the PCIN priority class, the signatures of one agent are classified as of high and low priority. To simplify the presentation and to match the characteristics of TM, we assume that only two signatures are allocated to one agent.

For the PCOUT priority class, we considered a fixed ratio of resources among the agents of different priorities during the entire execution of a program, but this ratio can be statically changed (before a program starts execution). For the PCIN priority class, the ratio of resources can be dependent on the identifier of the agent (thread) that issues an allocation.

We call $prioHigh$ ($prioLow$) the number that defines the high priority (low priority). Then, the ratio of resources assigned is given by the ratio of the priority values, that is

$$s\_factor = \frac{prioHigh}{prioLow} \tag{6.1}$$

To avoid any misunderstanding in the presentation, we indicate the priority class by adding the subscript "out" for the PCOUT class and "in" for the PCIN class to s_factor, $prioHigh$ and $prioLow$.

In an incoming allocation request, *FlexSig* tries to distribute the Bloom filters according to the different priorities and the corresponding ratios of resources (s_factor_{out} and s_factor_{in}). As in the symmetric case, a very accurate allocation algorithm leads to a very complex implementation that has a negligible improvement of performance compared with our implemented policy, which we explain below.

First, the controller computes the maximum number of Bloom filters that a low priority agent (excluding the new agent) may have:

$$nbL_{out} = \left\lfloor \frac{T}{\#agH \times s\_factor_{out} + \#agL} \right\rfloor \tag{6.2}$$

where $\#agH$ is the number of agents of high priority, and $\#agL$ is the number of agents of low priority (including the new agent). The controller also calculates the maximum number of Bloom filters that a high priority agent (excluding the new agent) may have:

$$nbH_{out} = \lfloor nbL_{out} \times s\_factor_{out} \rfloor \tag{6.3}$$

Once the controller has calculated the maximum number of Bloom filters per agent, it has to calculate how many of these Bloom filters correspond to the internal high priority signature and how many correspond to the internal low priority signature, and then it frees the Bloom filters of the signatures that exceed these values.

Particularly, for the resources of each agent, the controller frees the number of Bloom filters of the high (PCIN) priority signatures that exceeds the value

$$nbH_{in}(x) = \left\lfloor \frac{x \times prioHigh_{in}}{prioHigh_{in} + prioLow_{in}} \right\rfloor \tag{6.4}$$

with $x = nbL_{out}$ for low priority agents, and $x = nbH_{out}$ for high priority agents. Also, the controller frees the number of Bloom filters of the low (PCIN) priority signatures that exceeds the value

$$nbL_{in}(x) = x - nbH_{in}(x) \tag{6.5}$$

with $x = nbL_{out}$ for low priority agents and $x = nbH_{out}$ for high priority agents. That is, the low priority signature for each agent gets an upper bound of resources given by the difference between the number of Bloom filters of the agent and the number of Bloom filters assigned to the high priority signature.

Finally, all the free Bloom filters are assigned to the signatures of the new allocated agent. In case of a high priority agent, $nbH_{in}(nbH_{out})$ (see Equation (6.4)) Bloom filters are assigned to its high priority signature, and the remaining free Bloom filters are assigned to its low priority signature. In the same way, when the agent is of low priority, $nbH_{in}(nbL_{out})$ Bloom filters are assigned to its high priority signature, and the remaining free Bloom filters are assigned to its low priority signature.

Again, this policy favours the incoming agent, but this advantage is lost in the next allocation. Moreover, regarding the distribution of resources assigned to each agent, the policy favours the internal low priority signature, assuring a lower bound of resources allocated to it of at least one Bloom filter.

6.1.2 Asymmetric Algorithms for TM

FlexSig is not intended to work exclusively with TM, but it is illustrative to show how TM applications can take advantage of asymmetric policies. These policies can be adapted to other applications easily, as the concept of asymmetry is more general.

There are several ways to take advantage of the characteristics of a TM system with a specific transactional code. In the following subsections we show some alternatives that we evaluate later in Section 6.3, based on the simple implementation of the previous section.

In a TM system the agents are the transactions which need to allocate two signatures, one for the read set, and one for the write set. Therefore we may use each of the two levels of priority described in Section 6.1.1 to exploit the asymmetries between transactions and between the read and the write set signatures. The algorithm could be simplified by using one unified signature [35] for both read and write set, but we choose to use two conventional signatures to simulate a more complex scenario.

Asymmetric Read/Write Signatures Using PCIN Priority Class

Many TM systems use a read and a write signature to collect the read and the write sets of the transactions [147] [106] [95] [155]. Benchmark programs show that there is a significant variability in the relative sizes of the read and write sets. In many occasions the read set is bigger than the write set, but sometimes they are similar, or even the write set is the bigger. The characteristics of the read and write set totally depend on the benchmark. Therefore, the false positive rate also depends on the characteristics of the read and write set. Thus, it would be of interest to adapt the sizes of the read or write signatures to optimize the overall false positive rate.

To explore read/write asymmetry, the PCIN priority class is used. The values of priorities can be different for each transaction. Specifically, the controller calculates Equation (6.2) and (6.3) with $s\_factor_{out} = 1$ (same priority for all transactions),

$$nb_{out} = nbH_{out} = nbL_{out} = \left\lfloor \frac{T}{\#agents} \right\rfloor \tag{6.6}$$

where $\#agents$ is the total number of transactions that have signatures in *FlexSig* (including the new transaction).

Then, the controller frees the Bloom filters from the PCIN high priority signatures that exceed the value $nbH_{in}(nb_{out})$ (Equation (6.4) with $x = nb_{out}$), and it frees the Bloom filters of the low priority signatures that exceed the value $(nb_{out} - nbH_{in}(nb_{out}))$.

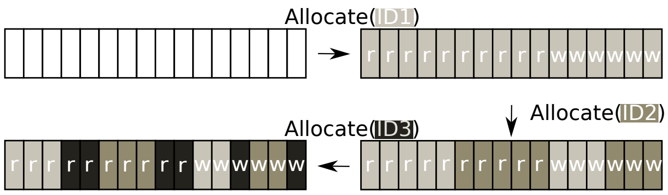


Figure 6.1: Example of an asymmetric R/W allocation algorithm with $s\_factor_{in} = 2$, $s\_factor_{out} = 1$ (no PCOUT) and with 3 signatures.

Table 6.1: Values calculated by *FlexSig* in the examples of the Figures 6.1, 6.2 and 6.3.

	Ex. Figure 6.1			*Ex. Figure 6.2*			*Ex. Figure 6.3*		
request	ID1	ID2	ID3	ID1	ID2	ID3	ID1	ID2	ID3
nbH_{out}	-	8	5	-	-	-	-	10	8
nbL_{out}	-	8	5	-	8	4	-	-	4
$nbH_{in}(nbH_{out})$	-	5	3	-	-	-	-	7	6
$nbL_{in}(nbH_{out})$	-	3	2	-	-	-	-	3	2
$nbH_{in}(nbL_{out})$	-	5	3	-	4	2	-	-	3
$nbL_{in}(nbL_{out})$	-	3	2	-	4	2	-	-	1
$freeBF$	16	8	6	16	8	8	16	6	4
$nbH_{in}(freeBF)$	10	5	4	8	4	4	12	4	3
$ID1\ resources$	10+6	5+3	3+2	8+8	4+4	2+2	12+4	7+3	6+2
$ID2\ resources$	-	5+3	3+2	-	4+4	2+2	-	4+2	3+1
$ID3\ resources$	-	-	4+2	-	-	4+4	-	-	3+1

Finally, all the free Bloom filters are assigned to the new transaction, with $nbH_{in}(nb_{out})$ Bloom filters for the high priority signature, and the remaining Bloom filters for the low priority signature.

Figure 6.1 shows an example of the algorithm with $T = 16$ and $s\_factor_{in} = 2$. The cells with an "r" are associated with the read signature (of high priority in this example) of the transaction, and the cells with a "w" are associated with the write signature (of low priority in this example).

Table 6.1 shows the values calculated by *FlexSig* for the example of Figure 6.1 (and for two more examples of next sections). From the row nbH_{out} until the row $nbL_{in}(nbL_{out})$ are values that are used to free exceeding Bloom filters in the signatures currently allocated in the module, the row $freeBF$ are the number of free Bloom filters that are finally assigned to the new thread, and the row $nbH_{in}(freeBF)$ indicates how many of these free Bloom filters are for the high priority signature. Furthermore, the rows *ID*1 *resources*, *ID*2 *resources* and *ID*3 *resources* indicate the number of BF assigned to the signatures of the corresponding thread after executing the allocation requests (decomposed in BF for the high priority signature + BF for the low priority signature).

At the beginning, all the Bloom filters of *FlexSig* are free. Then, the thread ID1 issues an allocation request, and *FlexSig* assigns all the resources to this thread, because it is the only one in the module. Having 16 free Bloom filters ($freeBF = 16$), *FlexSig* assigns $nbH_{in}(16) = 10$ Bloom filters to the read signature and the remaining Bloom filters to the write signature. For the second allocation request from thread ID2, *FlexSig* is not empty, and it has to free the exceeding Bloom filters from the signatures allocated in the module. For doing that, *FlexSig* calculates the value $nb_{out} = nbH_{out} = nbL_{out} = 8$ ($\#agents = 2$), and it frees the Bloom filters of the ID1 read signature that exceeds the value $nbH_{in}(8) = 5$ (5 Bloom filters are freed), and it frees the Bloom filters of the ID1 write signature that exceeds $nbL_{in}(8) = 3$ (3 Bloom filters are freed). After this, *FlexSig* assigns the 8 free Bloom filters ($freeBF = 8$) to the new ID2 thread ($nbH_{in}(8) = 5$ Bloom filters for the read signature and the remaining Bloom filters to the write signature). The last allocation request comes from thread ID3; *FlexSig* calculates $nb_{out} = nbH_{out} = nbL_{out} = 5$, ($nbH_{in}(5) = 3$) and ($nbL_{in}(5) = 2$), and it frees the exceeding Bloom filters from ID1 and ID2 signatures (2 exceeding Bloom filters for the read signatures and 1 for the write signatures). Finally, the 6 free Bloom filters ($freeBF = 6$) are assigned to the new ID3 thread, assigning $nbH_{in}(6) = 4$ Bloom filters to the read signature and the remaining Bloom filters to the write signature.

Asymmetric Signatures by Transaction Identifier Using PCOUT Priority Class

Benchmark programs have transactions with very different characteristics regarding the total size of the data sets and their access patterns. To take advantage of this, the allocation algorithm assigns resources based on the identifiers of the transactions using the PCOUT priority class. Thus, there is a set of more demanding transactions (that require bigger signatures to achieve a reasonable false positive rate), that are marked as high priority transactions, and others that are more lightweight, and that are marked as low priority transactions. The controller distributes the resources asymmetrically among transactions based on priorities, but continues distributing resources equally between read and write signatures (no PCIN priority class).

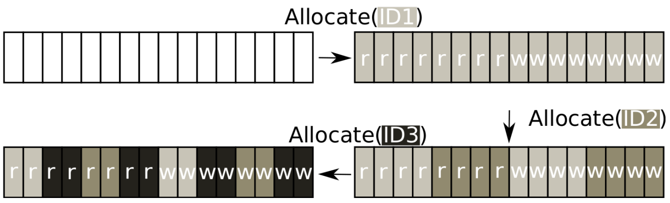

Allocate(ID1)
r r r r r r r r w w w w w w w w
Allocate(ID2)
r r r r r r r r w w w w w w w w
Allocate(ID3)
r r r r r r r r w w w w w w w w

Figure 6.2: Example of an asymmetric allocation algorithm based on transaction identifier with $s\_factor_{out} = 2$, $s\_factor_{in} = 1$ (no PCIN) and with 3 Transactions (ID1 and ID2 of low priority, and ID3 of high priority).

In a new allocation request, the controller uses a simplified version of Equations (6.4) and (6.5), and it frees the Bloom filters from the signatures (both read and write signatures) that exceed the value

$$nb_{in} = nbL_{in}(R) = nbH_{in}(R) = \left\lfloor \frac{R}{2} \right\rfloor \tag{6.7}$$

with $R = nbH_{out}$ (Equation (6.3)) in case of a high priority transaction, and $R = nbL_{out}$ (Equation (6.2)) in case of a low priority transaction.

Finally, the controller assigns nb_{in} Bloom filters to the write (read) signature of the new transaction, and the remaining free Bloom filters to the read (write) signature.

Figure 6.2 shows an example of this algorithm, with $T = 16$ and $s\_factor_{out} = 2$, and with three transactions, ID1 and ID2 with low priority, and ID3 with high priority. The values calculated by *FlexSig* are shown in Table 6.1, and the description of the algorithm is similar to the description done before for the example of Figure 6.1.

Combining PCOUT and PCIN Priority Classes

The two previous strategies of using asymmetry in Section 6.1.2 and Section 6.1.2 are totally orthogonal and compatible. The *FlexSig* controller can take advantage of priorities if a benchmark is asymmetric in read and write signatures (PCIN priority class), and at the same time has transactions with much bigger data sets than others (PCOUT priority class). In this case, the controller behaves exactly as explained in Section 6.1.1.

Figure 6.3 shows an example of this algorithm, with $s\_factor_{out} = 2$, $s\_factor_{in} = 3$ and $T = 16$, and with three transactions, ID1 with high PCOUT priority, and ID2 and ID3 with low PCOUT priority. In this example the write signature is of low PCIN priority. The values

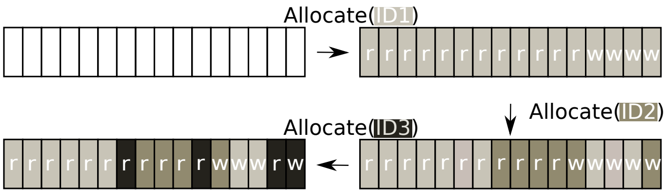

Allocate(ID1)
Allocate(ID2)
Allocate(ID3)
r r r r r r r r r r r r w w w w
r r r r r r r r r r r r w w w w
r r r r r r r r r r r w w w r w

Figure 6.3: Example of an asymmetric allocation algorithm combining PCOUT and PCIN priority classes, with $s\_factor_{out} = 2$ and $s\_factor_{in} = 3$ (ID1 of high PCOUT priority, ID2 and ID3 of low PCOUT priority; write signature of low PCIN priority).

calculated by *FlexSig* are shown in Table 6.1, and the description of the algorithm is similar to the description done before for the example of Figure 6.1.

6.2 Implementation Issues

In this section we briefly discuss two implementation issues: the placement of the module in the system and the hardware implementation of the module.

6.2.1 Placement of *FlexSig* in a Multicore Processor

There are several possibilities to place the module in the chip depending on the system, the target tool, the interconnection network, etc. In a system with a ring interconnection network, the *FlexSig* module could be placed in the cache controllers or connected to the network. In case of a system with a distributed directory cache coherence protocol, the module could be partitioned among directories based on address ranges, or the system could be clustered, by assigning one or more cores to a predefined directory. These two techniques could also be combined.

An example of location could be a centralized *FlexSig* in a ring-based TM system with a MESI cache coherence protocol. The module is connected to the on-chip interconnection network, physically associated with a switch of the ring. The scheme defined for the *FlexSig* module is very similar to the one defined for the Pacman module in Chapter 4.

6.2.2 Hardware Implementation in a TM System

This section describes the hardware implementation of the *FlexSig* in the context of a TM system. We propose a hardware implementation for asymmetric allocation algorithms using

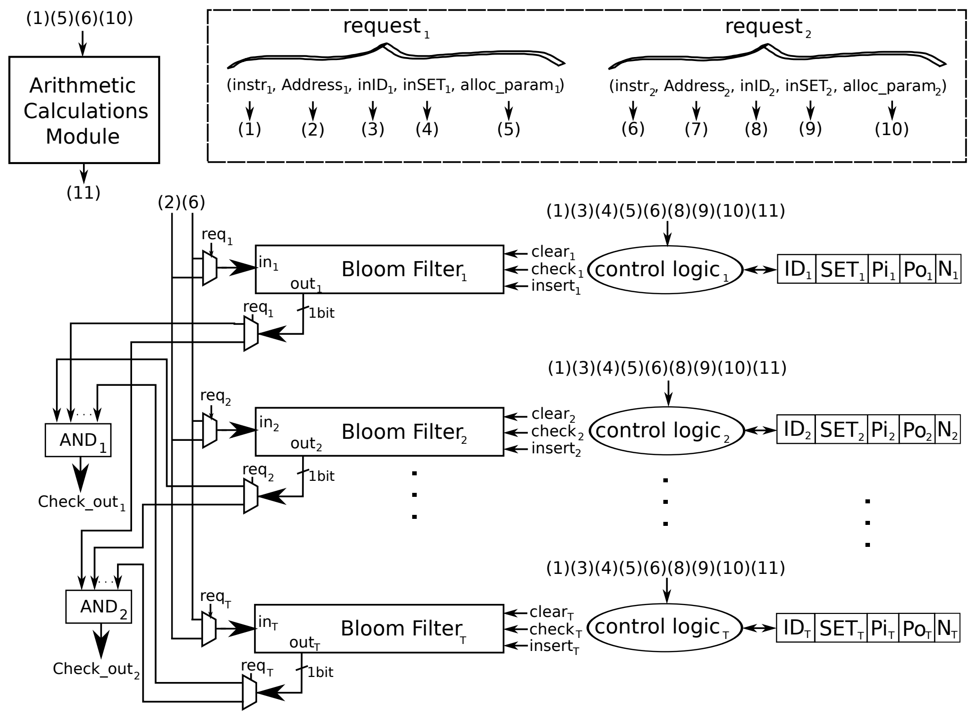


Figure 6.4: The basic *FlexSig* elements in a two-way implementation (issue 2 instructions).

read and write signatures. Despite the fact that this instance implementation is specifically designed for a TM system, it can be easily extended to other applications.

We assume a single transaction request to allocate the write and read signature. The hardware proposed implements the two classes of priorities described in Section 6.1.1. Symmetric allocations are performed by just setting the corresponding priorities to one (resulting in s_factor_{in} or s_factor_{out} equal to one).

6.2.3 Basic Elements

Figure 6.4 shows the basic elements and signals of a *FlexSig* module that can issue two instructions in parallel. The controller can be adapted to manage a greater number of parallel instructions at the expense of increasing complexity. Below, we describe the figure in detail.

There are two requests managed by the controller in parallel ($request_1$ and $request_2$ in the figure), each one composed of the code of the instruction ($instr_1$ and $instr_2$), an address ($address_1$ and $address_2$) used only in insert and check operations, the identifier of the transaction ($inID_1$ and $inID_2$), the identifier of the signature ($inSET_1$ and $inSET_2$) used only in insert and check operations and the allocation parameters ($alloc\_param_1$ and $alloc\_param_2$) used only in allocation operations. The allocation parameters include $prioHigh_{in}$, $prioLow_{in}$, $pHSig$, which indicates if the high priority signature is the read signature ($pHSig$ is one) or the write signature ($pHSig$ is zero), and $pOUT$, which indicates if the transaction is of high priority ($pOUT$ is set to one) or low priority ($pOUT$ is set to zero).

The controller has several simple elements that are replicated for each Bloom filter to achieve maximum parallelism:

– **Bloom_filter$_\mathbf{x}$**: it is a storage element composed of a register and a hash function.

– **in$_\mathbf{x}$**: input port for the address in the insert operation.

– **out$_\mathbf{x}$**: one bit output port for the result of the check operation.

– **ID$_\mathbf{x}$**: register to store the identifier of the transaction to which the $Bloom\_filter_x$ is allocated.

– **SET$_\mathbf{x}$**: register of one bit to identify a read ($SET_x = 1$) or write ($SET_x = 0$) signature.

– **Pi$_\mathbf{x}$**: one bit register that indicates that a Bloom Filter is part of a high priority signature or a low priority signature (PCIN priority).

– **Po$_\mathbf{x}$**: one bit register that indicates that the Bloom Filter is part of a high priority transaction or a low priority transaction (PCOUT priority).

– **N$_\mathbf{x}$**: register that stores the order of the Bloom filter in the signature. The N_x values are used to simplify the allocation algorithm (Section 6.2.5). For each signature, all the Bloom filters are numbered from one to the total number of Bloom filters in the signature, in consecutive and ascendant order.

– **control logic$_\mathbf{x}$**: is a simple per Bloom filter control logic that performs the Check, Insert, Allocate and Deallocate operations by generating the required control signals (see Section 6.2.5).

There are also several signals in each Bloom Filter that are activated by the **control logic**$_{\mathbf{x}}$ controller:

- **req**$_{\mathbf{x}}$: this signal is used to control input and output multiplexers for each Bloom filter.
- **clear**$_{\mathbf{x}}$: this signal is used for the allocating and deallocating request to clear the corresponding Bloom Filter.
- **check**$_{\mathbf{x}}$: is an enabling signal for reading the output of the Bloom filter.
- **insert**$_{\mathbf{x}}$: synchronous load signal for the Bloom filter.

Furthermore, the controller has an arithmetic calculations module that calculates the maximum number of Bloom filters per signature according to their priorities. These values are required by all the per filter control logic in each allocation request. Below, we describe this module in detail.

6.2.4 Allocation Implementation: Arithmetic Calculations

In this subsection we show the implementation of the arithmetic part of the allocation algorithm. We present an implementation for specific values of some parameters in order to fully consider all possible optimizations. The extension to other values requires the redesign of some parts in order to optimize them for the specific parameters. Specifically we present the design for T=64 (total number of Bloom filters in *FlexSig*), and priorities defined by three bits (prioHigh and prioLow are three bit numbers for both the PCIN and PCOUT priorities).

In our implementation, the high and low priority signatures correspond either to the read signature (when $pHSig$ is one) or to the write signature (when $pHSig$ is zero).

Figure 6.5 shows the generation of the allocation signal. If one of the two request is an Allocate, the controller executes the instruction in isolation, not serving additional requests until the end of the allocation, because this instruction can potentially affect all the Bloom filters. Furthermore, Figure 6.5 also shows how the number of transactions of high priority (*#agH*) and low priority (*#agL*) are maintained in a register and are updated in each allocate and deallocate by an up/down counter. The input parameters of the up/down counter are the type of the requests (references (1), (2) and (3) in the figure), and the priority of the requests (pOUT1 and pOUT2).

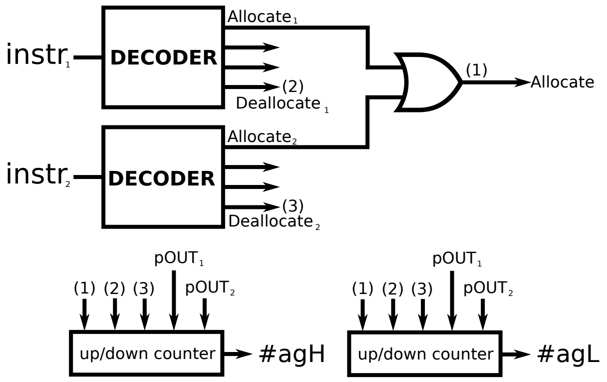


Figure 6.5: Generation of the allocation signal (common to all the Bloom Filters) and the update of the registers that count the number of transactions of low priority (*#agL*) and high priority (*#agH*) in the *FlexSig*.

The calculations necessary to perform the allocation are described by Equations (6.2), (6.3), (6.4) and (6.5). In order to reduce hardware complexity, and since our allocation algorithm is in fact heuristic, we designed the arithmetic units so that an error of one unit is allowed with respect to the exact arithmetic implementation of these equations. These errors have a negligible effect in the final results (as we will see in the evaluation in Section 6.3), but allow a significant hardware reduction.

Figure 6.6 shows the calculation of nbL_{out} (Equation (6.2)) and nbH_{out} (Equation (6.3)). First, it calculates the divisor in Equation (6.2) by multiplying s_factor_{out} by $\#agH$ and adding $\#agL$. At this point, according to Equation (6.2) a division operation is performed. The implementation of the division operation is simplified because: i) the dividend is a constant of the system (T); ii) the output is a six bit integer number (the range is from 1 to T, with T=64 in our implementation); and iii) we allow an error of one unit with respect to the exact arithmetic calculation. This allowed us to use a look-up table to implement the divisor (due to the reduced number of bits of the input).

Figure 6.6 shows the number of bits (integer + fractional) required at the input/out of each module so that a maximum of one unit of error is obtained in the final result with respect to the exact arithmetic implementation.

As we show in the figure only the leading five bits of the denominator are necessary at the input of the divisor unit (look-up table), therefore the result of the previous adder is

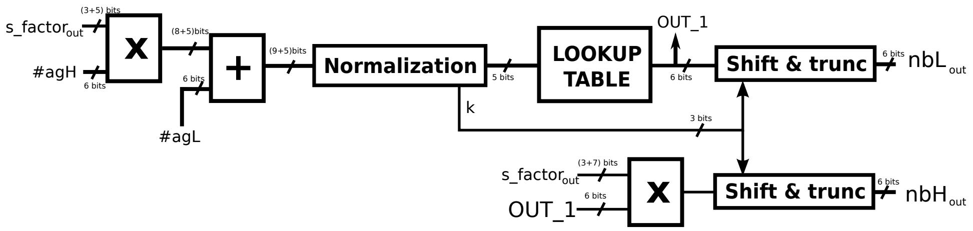


Figure 6.6: Calculation of the maximum number of Bloom filters for high and low priority transactions.

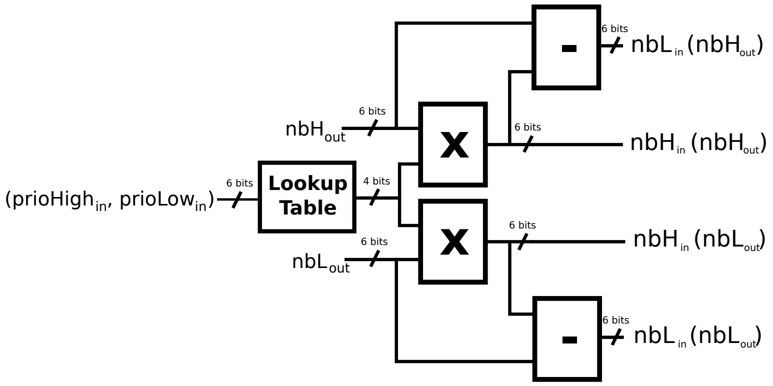


Figure 6.7: Calculation of the maximum number of Bloom filters per signature, depending on its priority and the priority of the transaction which it belongs.

normalized and truncated to five bits. The normalization is compensated at the output with the corresponding shift and a truncation to integer is performed.

To calculate nbH_{out} (also in Figure 6.6), the output of the divisor (OUT_1 in the figure) is multiplied by s_factor_{out}, and then, the result is shifted and truncated to an integer (as for nbL_{out}).

In Figure 6.7 we show the calculation of the maximum number of Bloom filters per high and low priority (for PCIN class) signatures, by distributing the values computed in Figure 6.6 (nbH_{out} and nbL_{out}) according to Equations (6.4) and (6.5), again allowing a maximum error of one unit. A look-up table is used to compute the result of the calculation $prioHigh_{in}/(prioHigh_{in} + prioLow_{in})$ (part of Equation (6.4)).

The output of the lookup table is a number in the interval (0,1], with four fractional bits. This value is multiplied by nbH_{out} and nbL_{out} to obtain the maximum resources of a high priority signature, $nbH_{in}(nbH_{out})$ and $nbH_{in}(nbL_{out})$. The maximum resources of a low priority signature are the remaining number of Bloom filters, which are computed by means of a subtraction operation (see Equation (6.5)).

6.2.5 Control Logic

In this section, we describe the control logic that generates local signals for each Bloom filter in Figure 6.4. Figure 6.8 shows the control logic for each one of the Bloom Filters of the *FlexSig*. In the figure BLOCK_1 and BLOCK_2 are used to activate $Insert_x$, $Check_x$ and $Deallocate_x$ instructions, and other intermediate signals used for the $Allocation$ instruction.

In BLOCK_1 we obtain the signals with reference (1), (2) and req_x. The signal req_x indicates, (when signal (1) is one), the request that matches the Bloom filter (zero if it matches $request_1$ and one if it matches $request_2$). BLOCK_2 generates $Check_x$, $Insert_x$ and $Deallocate_x$ signals, depending on signals (1) and (2).

If the instruction is an insert, the $Insert_x$ signal is generated (in BLOCK_2), and it enables directly the input of the Bloom filter (in_x in Figure 6.4). If the instruction is a Check, the $Check_x$ signal is generated (in BLOCK_2), and it enables directly the output (out_x in Figure 6.4) and the input (in_x in Figure 6.4) of the Bloom filter .

The $Deallocate_x$ signal produces the $clear_x$ signal that clears the register of the Bloom filter, and ID_x is also set to 0 (in BLOCK_3 of the figure).

BLOCK_4 is used for the allocation algorithm, and it is explained in the next subsection.

Allocation Request

The allocation request involves all the Bloom Filters, and therefore, this request can not be executed in parallel with other requests.

The allocation process is divided in two phases (controlled by a signal) that are needed to set all N_x values (that indicate the order of the Bloom Filter in the signature) of the new signatures, and to establish the corresponding values of SET_x, Pi_x and Po_x.

First phase: In the first phase (signal *phase* equal to zero), the controller frees the exceeding Bloom filters of the signatures. Each signature has its Bloom filters numbered in ascendant order, storing this number in N_x. The controller frees the Bloom filters that have a N_x value that exceeds the maximum number of Bloom filters allowed for that signature according to its priority (reference (6) in the BLOCK_3 of Figure 6.8). It also frees the Bloom filters with $ID_x = 0$ (reference (7) in BLOCK_3 of Figure 6.8). During this first phase of the

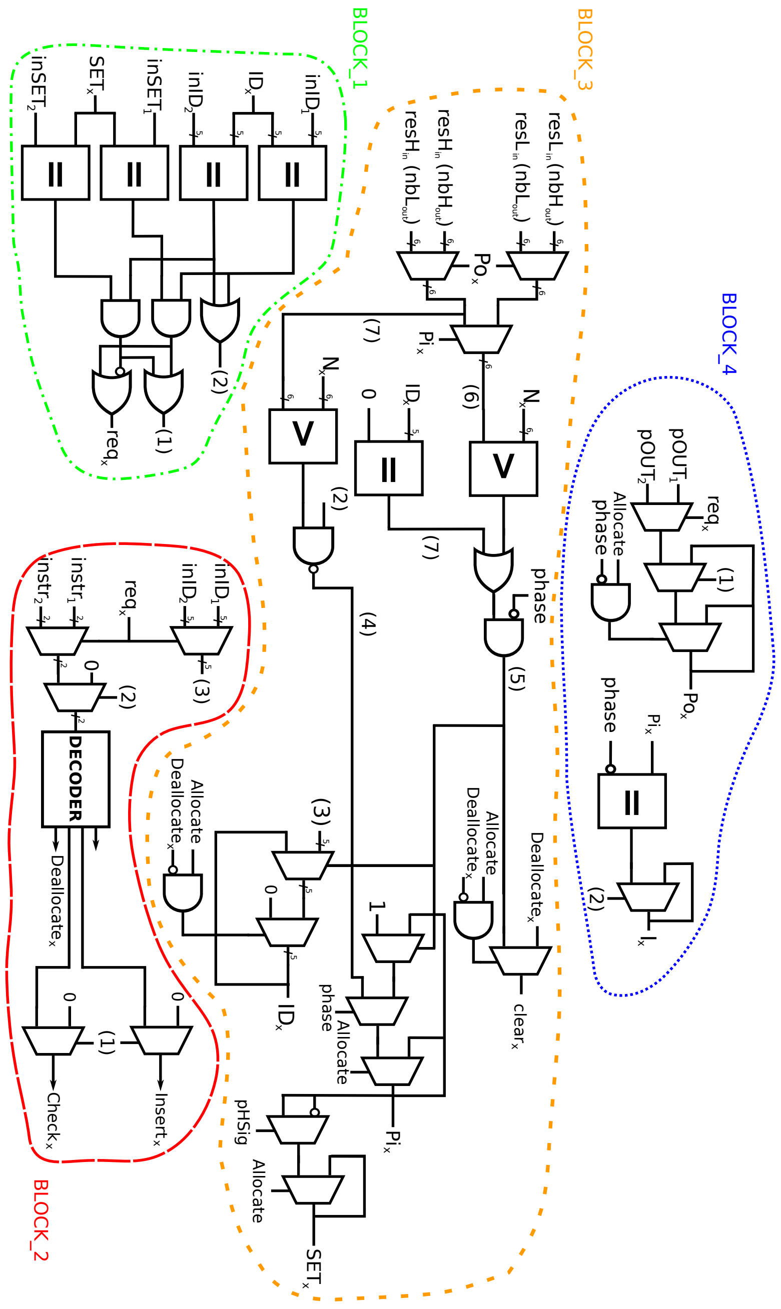


Figure 6.8: Control logic for each Bloom filter.

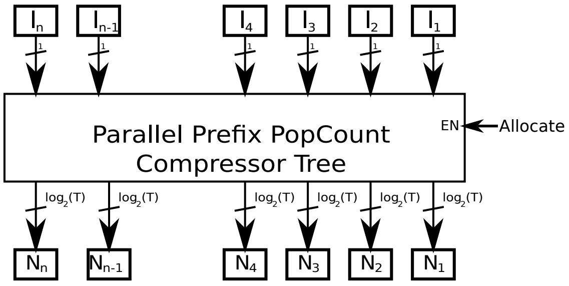


Figure 6.9: Generation of the $\mathbf{N_x}$ values with a parallel prefix popcount compressor tree.

allocation the corresponding Bloom filters are cleared (signal $clear_x$ in BLOCK_3), the transaction identifier is set (ID_x in BLOCK_3) to the input transaction identifier (either $inID_1$ or $inID_2$, reference (3) in BLOCK_2), the priority of the signature (Pi_x) is set to one (BLOCK_3 of the figure) and the signature type (SET_x) is set to zero (for the read signature) or one (for the write signature), depending on $pHSig$ in BLOCK_3. Note that all the Bloom Filters of the new transaction have high priority in this first phase.

Additionally, the N_x values for the Bloom filters of the new transaction are determined (for the high priority signature). Figure 6.9 shows how this computation is performed. The signals I_x are set to one if the corresponding Bloom filter belongs to the new transaction (if the signal with reference (2) is active in BLOCK_4 of Figure 6.8) and if it belongs to a high priority signature (as are all the Bloom filters of the new transaction in this first phase). The values of N_x are calculated from the I_x values with the parallel prefix population count compressor tree show in Figure 6.9, that computes

$$N_x = \sum_{j=1}^{x} I_j \tag{6.8}$$

with $1 \leq x \leq T$. Note that this circuit is composed of full adders organized in the form of a prefix tree.

Furthermore, in this first phase, the priority of the transaction is established by setting Po_x. The controller assigns to Po_x the value of $pOUT_1$ or $pOUT_2$ (shown in BLOCK_4 of Figure 6.8).

Second phase: At the end of the first phase, there is only a signature (the high priority signature). In the second phase (*phase* equal to one), the controller assigns Bloom filters to the low priority signature. This action is taken by the controller by setting Pi_x to zero (reference

(4) in BLOCK_3 of Figure 6.8) to those Bloom filters of the high priority signature whose N_x values exceeds the maximum value previously calculated for the high priority signatures (reference (4) in Figure 6.8).

The low priority signature also needs to set the correct N_x values. In this phase, the I_x values are set to one if the Bloom filter belongs to the low priority signature (Pi_x is zero) of the new transaction (BLOCK_4 in Figure 6.8). The new N_x values for the low priority signature are calculated as in the first phase (Figure 6.9), reusing the same circuit. Moreover, the signature type (SET_x) is set depending on $pHSig$, as in the first phase.

Note that the control logic per Bloom filter (Figure 6.8) is composed of simple gates, and that some datapath elements are at most six bits wide. In general those datapath elements have bit widths that grow logarithmically with the total number of Bloom filters in *FlexSig*. Moreover, the hardware for calculations of Figure 6.7 and 6.6 are shared among all the Bloom filters, and its datapath elements have widths that grow also logarithmically with the total number of Bloom filters. Therefore, the hardware complexity to support different priorities in *FlexSig* scales in a reasonable way (logarithmically) with the total size of *FlexSig*.

6.3 Evaluation for a TM System

In this section we evaluate the asymmetric policies in *FlexSig* in the context of a TM system. Specifically we use the information regarding the read and write sets from some well known TM benchmarks. Our goal is to evaluate the improvements in the false positive rate by implementing asymmetric strategies in *FlexSig*. The false positive rate is directly related with performance [182], since a false positive leads to an unnecessary conflict, and therefore one of the conflicting transactions has to stall or rollback and restart, with the consequent inefficiency.

6.3.1 Experimental Setup

We use RSTM [100] as TM system, and PIN [81] to instrument the transactional code (to gather read and write sets) and to simulate hardware signatures.

For testing our scheme, we chose some widely accepted benchmarks for testing on TM systems: STAMP [25] benchmarks, some micro benchmarks included in RSTM and "Eigen-Bench" benchmark [73] (a simple synthetic benchmark that can be configured to stress different TM characteristics). Table 6.2 shows the inputs of these benchmarks in our experiments.

Table 6.2: Benchmark Inputs.

Benchmark	input
Intruder	-a10 -l16 -n4096 -s1
Vacation-high	-n4 -q60 -u90 -r1048576 -t4096
Vacation-low	-n2 -q90 -u98 -r1048576 -t4096
Yada	-a10 -i ttimeu10000.2
Bayes	
Genome	-g128
Kmeans-high	-m15 -n15 -t0.05 -i random-n2048-d16-c16.txt
Kmeans-low	-m40 -n40 -t0.05 -i random-n2048-d16-c16.txt
Labyrinth	-i random-x256-y256-z3-n256.txt
Ssca2	-s14 -i1.0 -u1.0 -l9 -p9
MicroBench	-X(5000/num_threads)
Eigen1	ID1=(RS:15,WS:15) ID2=(RS:30,WS:2)
Eigen2	ID1=(RS:30,WS:2) ID2=(RS:4,WS:1)
Eigen3	ID1=(RS:75,WS:14)

Table 6.3: Size of the read and write sets for different benchmarks.

	Intruder	Vac.-high	Vac.-low	Yada	List	DList	Bayes	Genome	Km.-high	Km.-low	Labyrinth	Ssca2	Hash	Tree	TreeOver.	Forest
RS	22	385	283	142	149	148	12.9	40.3	13	13	92	1	2	17.6	39.7	17.6
WS	2.6	8.9	6.4	21	0.3	0.6	3.2	0.03	13	13	88	2	0.3	2	6	2

Table 6.3 shows the average size of the read and write sets of the benchmarks used in the evaluation in Section 6.3.2.

Table 6.4 shows the four different configurations that we use to achieve the adequate false positive rates to compare the improvements in all the cases (more details below).

6.3.2 Evaluation for the PCIN Priority Class

In this case there is no priority for distribution of resources among transactions, but for each transaction a priority is provided to distribute the resources for the read and write set (PCIN priority class).

For testing this experiment, we chose STAMP [25] benchmarks and some micro benchmarks included in RSTM. We simulate the benchmarks using from 2 to 16 threads in a sys-

Table 6.4: Configuration of the signatures used.

	Conventional	*FlexSig*	Benchmarks
Conf. 1	32sigs x (256bits,k=4)	128BF x 64bits	Intruder, Vacation, Yada, List, DList
Conf. 2	32sigs x (64bits,k=4)	128BF x 16bits	Bayes, Genome, Kmeans, Labyrinth, Ssca2, Hash, Tree, TreeOver., Forest
Conf. 3	-	128BF x 8bits	Eigen1, Eigen2
Conf. 4	(a) 256sigsx(64bits,k=4) (b) 128sigsx(64bits,k=4) (c) 64sigsx(64bits,k=4) (d) 32sigsx(64bits,k=4)	(a) 1024BFx16bits (b) 512BFx16bits (c) 256BFx16bits (d) 128BFx16bits	Eigen3

tem with 32 conventional signatures (parallel Bloom filters), and in a system with a *FlexSig* module. In this experiment we use the Conf. 1 and Conf. 2 shown in Table 6.4: The first configuration uses 32 signatures (16 read signatures and 16 write signatures) of 256 bits and $k = 4$ (k is the number of hashes in the parallel Bloom filter) for conventional signatures, and a *FlexSig* with 128 Bloom filters of 64 bits (the same total number of bits as conventional signatures). This configuration is used for the evaluation with the benchmarks that produce a higher rate of false positives. The second configuration uses 32 signatures of 64 bits and $k = 4$ for conventional signatures, and a *FlexSig* with 128 Bloom Filters of 16 bits (the same total number of bits than conventional signatures). It is used for the evaluation for benchmarks with lower levels of false positives.

For reference, Figure 6.10 shows the percentage of reduction on false positives of *FlexSig* with no priorities ($s\_factor_{in} = 1$ and $s\_factor_{out} = 1$) with respect to conventional Bloom filters (higher is better, and a 100% reduction means no false positives). The Table 6.4 shows the configuration of the signatures used for this experiment. In contrast to the results showed in Chapter 5, this simulation corresponds to an implementation with separate read and write set signatures, and the number of total resources also differs in both cases. As it was shown in Chapter 5, there is a significant reduction in the number of false positives in this case (the reduction improves as the number of threads is reduced) due to the flexibility in resource management.

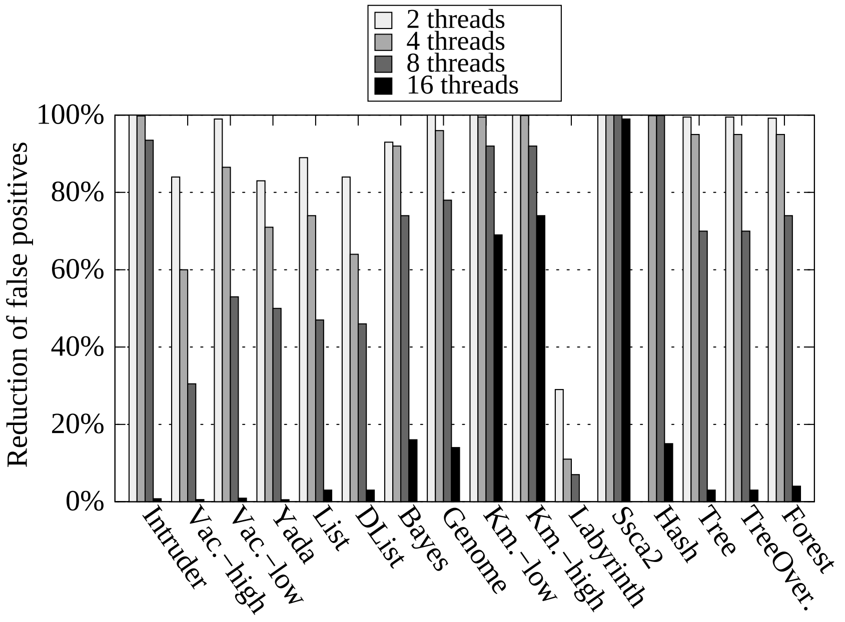


Figure 6.10: Percentage of decrease of false positives in symmetric *FlexSig* compared with conventional signatures.

Table 6.5: s_factor_{in} $(prioHigh_{in}|prioLow_{in})$ for 2, 4, 8 and 16 threads.

	Intruder	Vac.-high	Vac.-low	Yada	List	DList	Bayes	Genome	Km.-high	Km.-low	Labyrinth	Ssca2	Hash	Tree	TreeOver.	Forest
2th.	1|1	7|1	7|1	3|1	7|1	7|1	4|1	1|1	1|1	1|1	7|6	1|1	1|1	2|1	2|1	2|1
4th.	3|1	7|1	7|1	1|1	7|1	7|1	3|1	1|1	5|4	5|4	7|6	1|1	1|1	3|2	2|1	3|2
8th.	2|1	4|1	6|1	1|1	6|1	6|1	2|1	1|1	4|3	7|6	1|1	1|1	1|1	3|2	4|3	3|2
16th.	2|1	2|1	2|1	2|1	6|1	6|1	5|4	4|3	1|1	1|1	1|1	1|1	4|3	4|3	3|2	3|2

Table 6.5 shows the values used for $prioHigh_{in}$ and $prioLow_{in}$ (that define s_factor_{in}) for the simulation using the PCIN priority class. We show the two values in each benchmark depending on the number of threads. We determined these values empirically, by profiling each benchmark for the different number of threads. In the table, white (shaded) cells indicate the cases where the read (write) signature is of high priority. When $prioHigh_{in}$ and $prioLow_{in}$ have the same value, it is represented also with a white cell, despite there is no priority. For

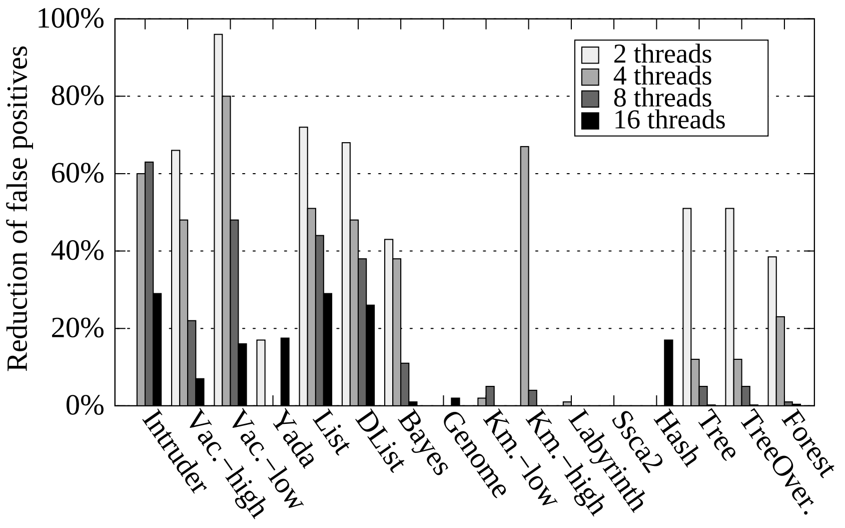


Figure 6.11: Percentage of decrease on false positives in *FlexSig* with asymmetric allocation policies (PCIN priority class) compared with the conventional *FlexSig*.

example, "Yada" benchmark with 2 threads uses the values $prioHigh_{in} = 3$ and $prioLow_{in} = 1$ (the read signature has high priority), the "Kmeans-high" benchmark with 4 threads uses the values $prioHigh_{in} = 5$ and $prioLow_{in} = 4$ (the write signature has high priority in this case), and the "Genome" benchmark with 2 threads uses the values $prioHigh_{in} = 1$ and $prioLow_{in} = 1$ (read and write signatures have the same priority).

Figure 6.11 shows the percentage of reduction in the number of false positives in a *FlexSig* when we use an asymmetric policy to implement priorities for the PCIN class, compared with a conventional *FlexSig* system implementing a symmetric policy. As shown in Figure 6.11, for some benchmarks the asymmetric policy in allocation of resources for the read and write set allows a further significant reduction in the number of false positives. For instance, "Intruder", "Vacation-high", "Vacation-low", "List", "DList", "Kmeans-high", "Tree" and "TreeOverwrite" achieve reductions higher than 50%. Other benchmarks (such as "Ssca2", "Labyrinth" or "Genome") do not get a significant advantage from using the asymmetric policy, so they can be executed using a symmetric allocation algorithm. The results are usually worse when

Table 6.6: Priorities used for single and multiple PCIN priorities.

	2 th.	4 th.	8 th.	16 th.
s_factor_{in}	3|2	6|5	6|5	6|5
$s\_factor_{m\_in}$	1|1 (ID1)	2|1 (ID1)	2|1 (ID1)	2|1 (ID1)
	7|1 (ID2)	7|1 (ID2)	6|1 (ID2)	3|2 (ID2)

the number of threads increases because the average signature size decreases leading to more saturated signatures that reduce the advantage of the asymmetry.

6.3.3 Evaluation for Multiple PCIN Priority Classes

The difference with the experiment in Section 6.3.2 is that in this case we evaluate the results for a configuration with a specific per transaction s_factor_{in} (PCIN priorities may differ among transactions).

The configuration used in *FlexSig* is the Conf. 3 in Table 6.4: a *FlexSig* with 128 Bloom filters of 8 bits. For this experiment we use "EigenBench" benchmark [73], with the Eigen1 configuration in Table 6.2. This benchmark has two different transactions, ID1 and ID2. ID1 has the same read and write set size (RS= 15, WS=15), and ID2 is configured with a much bigger read set, larger than the write set (RS= 30, WS=2). Half of the threads execute ID1 transactions, and the other half execute ID2 transactions.

Table 6.6 shows the values of $prioHigh_{in}$ and $prioLow_{in}$ used in this experiment. We show the values for the case when a single PCIN priority is used for the two transactions (row s_factor_{in}), and when a different PCIN priority is used (row $s\_factor_{m\_in}$). For example, with 2 threads, the s_factor_{in} is defined by $prioHigh_{in} = 3$ and $prioLow_{in} = 2$, and the $s\_factor_{m\_in}$ is defined by $prioHigh_{in} = 1$ and $prioLow_{in} = 1$ for the ID1 transaction, and it is defined by $prioHigh_{in} = 7$ and $prioLow_{in} = 1$ for the ID2 transaction. The white cells indicate that the read signature has high priority, or that both signatures have the same priority (when $prioHigh_{in} = prioLow_{in}$), and the shadow cells indicate that the write signature has high priority.

Figure 6.12 shows the results comparing the two configurations. The use of a per transaction priority allows a further significant reductions in the number of false positives. Therefore, as we show with this synthetic benchmark, there might be cases where it is worth to use a per transaction PCIN priority.

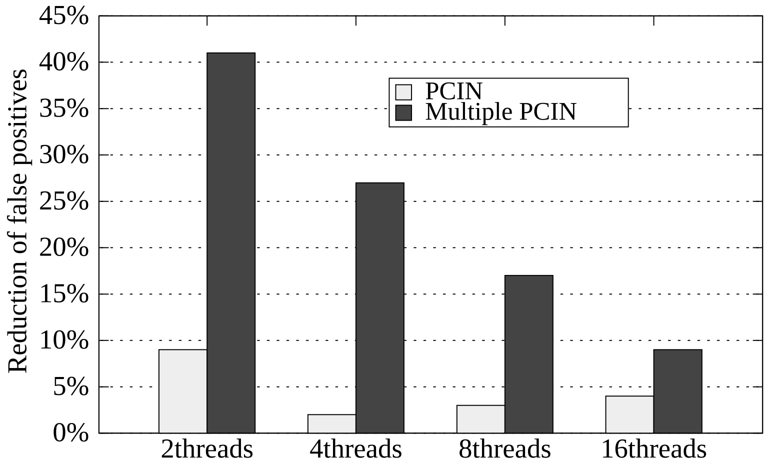


Figure 6.12: Percentage of reduction of false positives for single PCIN priority and for Multiple (one per transaction) PCIN priorities (compared with a symmetric *FlexSig*).

6.3.4 Evaluation for Combined PCOUT and PCIN Priority Classes

In this experiment, we explore the two priority classes (PCOUT and PCIN) using also "Eigen-Bench" benchmark. Specifically we configure this benchmark according to Eigen2 configuration in Table 6.2. As before we use two types of transactions. The first type (ID1) is a transaction with a read set of 30 addresses, and a write set of 2 addresses, and the second type (ID2) has a read set of 4 addresses and a write set of one address. Half of the threads execute transactions ID1, and the other half execute ID2. We have two classes of asymmetry, one because of the different size of both types of transactions (PCOUT priority class), and the other because the read set is much bigger than the write set in both types of transaction (PCIN priority class).

The *FlexSig* setup for this experiment is the Conf. 3 in Table 6.4 (128 Bloom filters of 8 bits). Furthermore, we use three priority configurations. The first configuration (PCIN) uses different priorities for the read and write signatures (as in the previous experiment), the second configuration (PCOUT) uses different priorities for different transactions depending in their IDs, and the third configuration (PCIN + PCOUT) uses a combination of the two previous configurations. Table 6.7 shows the different s_factor values used in the three configurations (represented in the form $prioHigh_{in}|prioLow_{in}$). The s_factor_{in} row shows the priorities of the read and write signatures (PCIN priority class), having the read signatures high priority

Table 6.7: s_factor used to evaluate both PCOUT and PCIN priority classes.

	2 th.	4 th.	8 th.	16 th.
s_factor_{in}	4|1	4|1	4|1	2|1
s_factor_{out}	3|1	3|1	2|1	2|1
s_factor_{in+out}	3|1 (out) 4|1 (in)	3|1 (out) 4|1 (in)	2|1 (out) 4|1 (in)	2|1 (out) 2|1 (in)

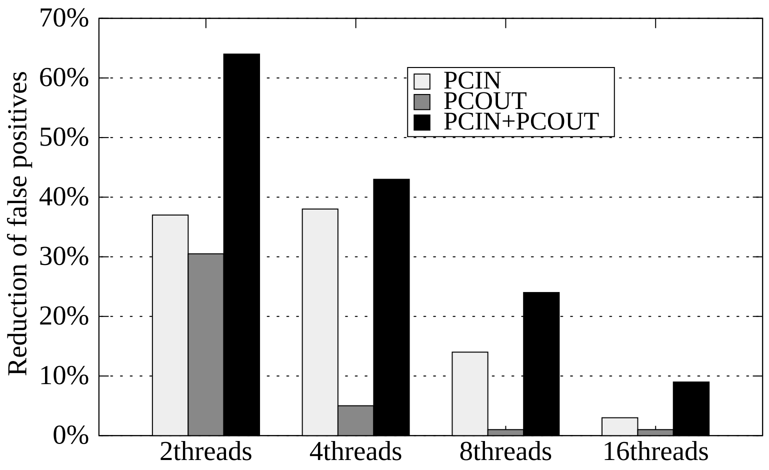


Figure 6.13: Percentage of decrease on false positives in *FlexSig* implementing priorities for PCIN priority class, for PCOUT priority class, and combining both PCIN and PCOUT priority classes.

in all cases; the s_factor_{out} row shows the priorities of the transactions depending on its identifier (PCOUT priority class); and the row s_factor_{in+out} shows the priorities for the combined priorities (PCOUT and PCIN priority classes at the same time).

Figure 6.13 shows the results of the three different configurations compared with symmetric *FlexSig*. It is clear that combining both kind of priorities the results are improved significantly. For instance, with two threads, the reduction in the number of false positives is under 40% when PCIN or PCOUT priorities are used alone. When both priorities are combined, the net effect is a reduction of about 65%.

Table 6.8: s_factor_{in} used to evaluate the scalability of *FlexSig*.

	2 th.	4 th.	8 th.	16 th.	32 th.	64 th.	128 th.
16-thr. system	3|1	2|1	3|2	4|3	-	-	-
32-thr. system	4|1	3|1	2|1	2|1	6|5	-	-
64-thr. system	5|1	5|1	5|1	3|1	3|2	4|3	-
128-thr. system	4|1	4|1	4|1	3|1	2|1	5|3	5|3

6.3.5 *FlexSig* with a High Number of Threads

In the same way that for the symmetric *FlexSig* described in Chapter 5, the results of asymmetric *FlexSig* are worse when the number of threads increases. In this section we demonstrate that this is not due to any scalability issue.

In this experiment we show that *FlexSig* can scale up to a high number of threads. The Conf. 4 of Table 6.4 is used to simulate an environment with a maximum number of 16, 32, 64 and 128 threads (configurations (a),(b),(c) and (d) respectively). We use the "EigenBench" benchmark with the Eigen3 configuration in Table 6.2, *FlexSig* implementing the PCIN priority class, and the s_factor_{in} values used are shown in Table 6.8 (in each cell, to the left is $prioHigh_{in}$ and to the right is $prioLow_{in}$).

Figure 6.14 shows the reduction of false positives of asymmetric *FlexSig* (PCIN priority class) compared with conventional Bloom filters for a system with a maximum of 16, 32, 64 and 128 threads. *FlexSig* achieves significant reductions, specially when the benchmarks are executed with less threads than the maximum allowed in the system. With this experiment we show that *FlexSig* is getting worse when it runs with the maximum number of threads allowed in the system, but it has good scalability, as the behavior has the same pattern with systems up to 128 threads. The worst results are when a benchmark is executed with the maximum number of threads in the system. However, even in the worst case, *FlexSig* clearly outperforms conventional Bloom filters.

6.3.6 Signature Size Comparison

This experiment compares the size of asymmetric *FlexSig* (PCIN priority class) with the size of conventional Bloom signatures. The idea is to build the two signature schemes in such a way that the false positive rate of both are roughly the same, and compare the sizes of their registers to achieve this rate. For simplicity, we chose the "EigenBench" benchmark with the

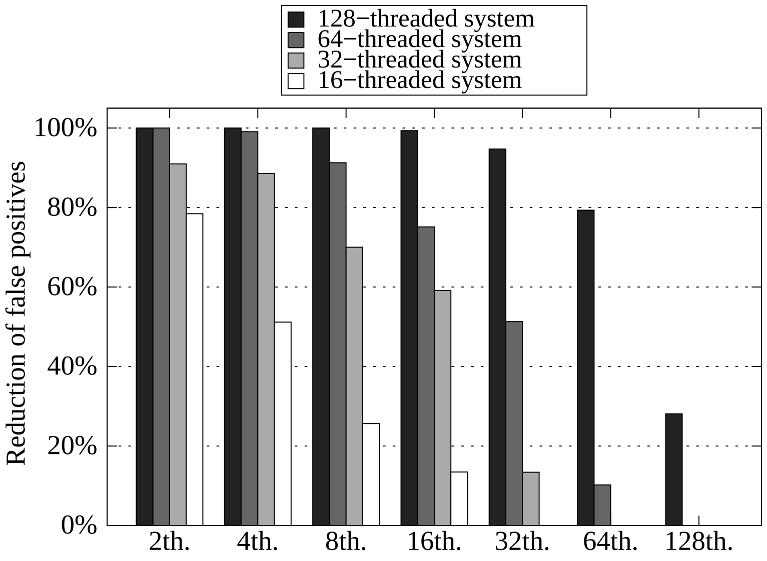


Figure 6.14: Percentage of decrease of false positives in asymmetric *FlexSig* (PCIN priority class) compared with conventional Bloom filters for "EigenBench" benchmark with up to 128 threads.

Table 6.9: Configuration of *FlexSig* and Bloom signatures.

	FlexSig			*Bloom*		
	#BF	BF size	prio	#sig	sig. size	k
2th.	128	16	4|1	32	208	4
4th.	128	21	3|1	32	224	4
8th.	128	30	2|1	32	224	4
16th.	128	51	5|3	32	224	4

Eigen3 configuration (Table 6.2) to do this comparison, and the false positive rate achieved in all the cases is bounded between $\mathbf{0.9}\%$ and $\mathbf{1.1}\%$.

The parameters used to configure the signatures are in Table 6.9. *FlexSig* is configured in all the cases with 128 Bloom filters (#BF), and the register size of the Bloom filter (BF size) and the PCIN priority (prio) change depending on the number of threads. Moreover, each one of the 32 Bloom signatures (16 signatures for the read set and 16 signatures for the write set) is composed by 208 bits (for 2 threads) or 224 bits (for 4,8 and 16 threads), and $k = 4$ in all the cases.

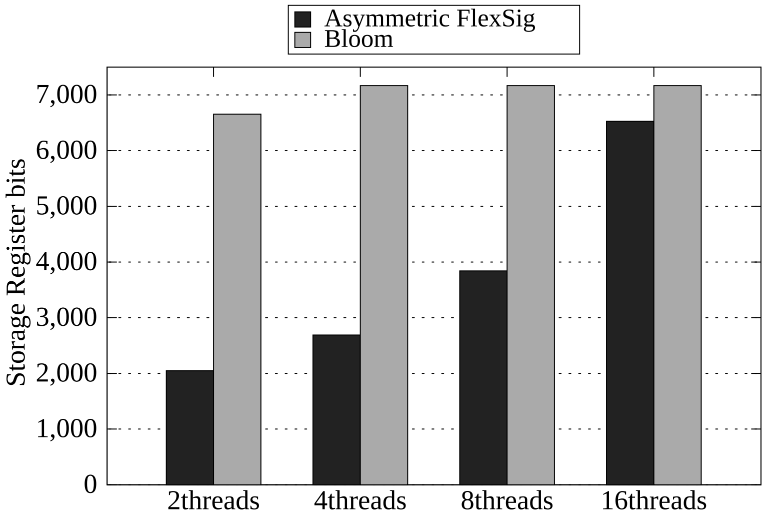


Figure 6.15: Number of bits required for registers in asymmetric *FlexSig* (PCIN priority class) and Bloom signatures, with a false positive rate between 0.9% and 1.1%.

Figure 6.15 shows the number of bits of storage needed for obtaining roughly the same false positive rate (the smaller the bars, the better) in each of the two compared signature schemes. We see that asymmetric *FlexSig* achieves very good results for 2, 4 and 8 threads, with register size reductions of the order of two and three times. Even with 16 threads, asymmetric *FlexSig* needs less bits than Bloom signatures.

6.4 Related Work

Quislant et al. [136] have proposed a new reconfigurable asymmetric signature (ASYM) to deal with the asymmetry of the read and write sets in TM systems. The high level idea is to configure the ASYM signature to establish the number of Bloom filters devoted to the read and the write sets (see Section 1.5.4).

This signature would be equivalent to a *FlexSig* module managing only one transaction and implementing the PCIN priority class. Because of the simplicity of the scenario, this module would not require an allocation algorithm, and the size of the signatures would be constant from the beginning to the end of the transaction (in *FlexSig* the size can be reduced in a running transaction). Compared with *FlexSig*, the scheme proposed in [136] is more limited, in the sense that it only deals with asymmetry of read and write sets of one transaction, while *FlexSig* is a shared module, which deals with many transactions and more types of asymmetries, such as asymmetry among user transactions or asymmetry among applications. Also, *FlexSig* is not intended to work only with TM, and it can be easily adapted to work with other applications or tools. In Section 6.4.1 we show the results of an experiment comparing asymmetric *FlexSig* with ASYM signatures. These results show that asymmetric *FlexSig* allows significant reductions in the number of false positives compared with ASYM signatures.

Korgaonkar et al. [83] propose to distribute the resources in *FlexSig* according to the size of the transactions. However, they do not propose any hardware implementation, nor do they describe the algorithm implemented in detail. In contrast, we propose a hardware approach of a general asymmetric algorithm (not only for TM) with several levels of asymmetry.

Scalable Bloom Filters [4], AdaptSig [129] or Dynamic Bloom Filters [61] propose alternatives in the same way: they expand signatures with more resources when the false positive rate reaches a prefixed level. For example, the Scalable Bloom Filters (SBF) are composed by one or more single Bloom filters; when the filters reach the fill ratio, a new filter is added to the SBF. Each successive Bloom filter is created with a tighter maximum error probability on a geometric progression, so that the compounded probability over the whole series converges to some predefined value. The check and insert operations are made by testing/setting all the filters. With this method, the false positive rate is always contained. However, it might be very difficult to implement these approaches in hardware, because an indefinite number of resources would be needed (it depends on the application).

6.4.1 Comparing Asymmetric *FlexSig* and ASYM signatures

We repeat the experiment of Section 6.3.2 for some benchmarks, but this time comparing the results of asymmetric *FlexSig* (with PCIN priority class) with the results of ASYM and conventional Bloom signatures. The configuration used for *FlexSig* is the Conf. 1 in Table 6.4 and the priorities of the Table 6.5. The system with ASYM signatures is configured with 16 ASYM signatures, each one composed by 8 Bloom filters (each Bloom filter composed by a H3 hash and a 64 bit register), that accumulate the same storage capacity that our *FlexSig*. The 8 Bloom filters of each ASYM signatures are distributed among the read and write set according to Table 6.10, where, in each cell, to the left it is the number of Bloom filters assigned to the read set, and to the right the number of Bloom filters assigned to the write set.

Table 6.10: Number of Bloom filters for read and write set in ASYM signatures composed by 8 Bloom filters (to the left the number of read Bloom filters and to the right the number of write Bloom filters).

	2threads	4threads	8threads	16threads
Intruder	4|4	5|3	5|3	5|3
Vacation-high	5|3	5|3	5|3	5|3
Vacation-low	6|2	5|3	5|3	5|3
Yada	3|5	3|5	3|5	3|5
List	6|2	6|2	6|2	6|2
DList	6|2	6|2	6|2	6|2

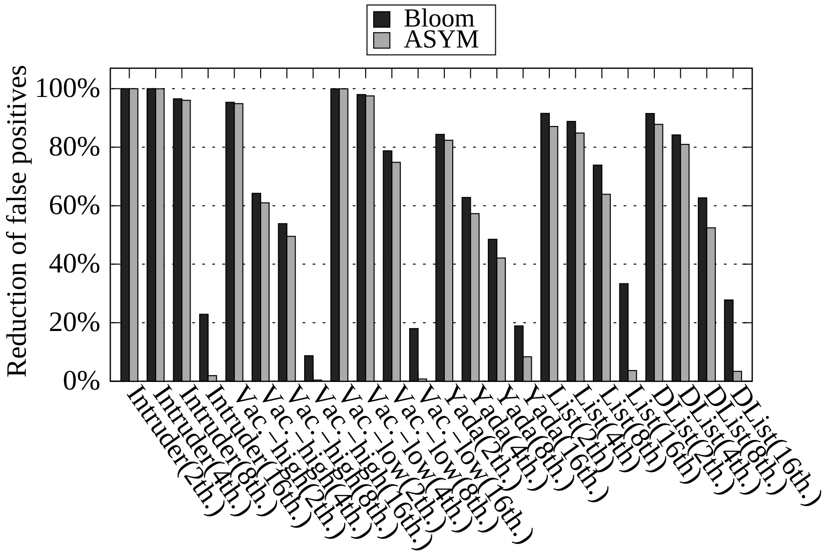


Figure 6.16: Percentage of decrease of false positives in asymmetric *FlexSig* implementing the PCIN priority class, compared with Bloom and ASYM signatures for "Intruder", "Vacation", "Yada", "List" and "Dlist" with 2,4,8 and 16 threads.

Finally, the conventional Bloom signature system is configured with 32 parallel Bloom filters, each one with a total size of 256 bits and 4 H3-hash functions.

Figure 6.16 shows the results of comparing ASYM and Bloom signatures with asymmetric *FlexSig* implementing the PCIN priority class (lower bars are better). We can see that, for 2 threads, the reduction of false positives in *FlexSig* is very significant in comparison with both ASYM and conventional signatures, because these signatures do not take advantage of all resources when the number of threads is less than 16. For this same reason, the better results of ASYM with respect to Bloom signatures are masked by the superiority of *FlexSig*. Just when the number of threads is 16 we can clearly appreciate the advantage of ASYM with respect to Bloom signatures. However, *FlexSig* has a clear advantage over the two alternatives in almost all the cases.

For this test, we only use a subset of the benchmarks that better represent the advantage of ASYM signatures over conventional Bloom filters. For the test used in Section 6.3.2 that are not shown in Figure 6.16, the better results of ASYM are obtained with a symmetric configuration, and therefore the results for the ASYM signatures are the same than for the conventional Bloom filters shown in Section 6.4.1, that are also clearly outperformed by asymmetric *FlexSig*.

6.5 Conclusion

FlexSig is an interesting approach for making signatures flexible and adaptable to different situations. However, the symmetric algorithm for allocating signatures proposed in Chapter 5 is very simplistic. In this chapter we explored more involved techniques to allocate signatures in *FlexSig* based on the asymmetry of the demands, and defined priorities for assigning more or fewer resources to the signatures. By implementing these asymmetric techniques we are able to significantly reduce the number of false positives compared with the symmetric *FlexSig* design in Chapter 5. We demonstrate with our experimental setup for a specific application (TM) that in some cases the reduction of false positives can be important, achieving reductions of over 60% in several benchmarks when implementing the PCIN priority class. We also evaluate other asymmetric policy with two priority classes that achieves reductions of up to 64%, and we perform a size comparison that demonstrates that asymmetric *FlexSig* can achieve up to a 60% reduction in the area occupied by the signature registers for a given upper bound in the false positive rate. Finally, a highly parallel high throughput hardware implementation has been developed, which allows a high performance asymmetric *FlexSig* module for future multicore systems.

These asymmetric policies strengthen the case for *FlexSig* as a flexible hardware resource to be used in a general purpose multicore processor, extending the original concept of flexibility in *FlexSig* with the adaptation to application-dependent characteristics (asymmetry).

Conclusions and Future Work

This dissertation presents novelty mechanisms related with hardware signatures in the context of multicore processors. Specifically, we focus on two mechanisms to help parallel programming in multicore processors (a HTM system and a Data Race Detector), and in a flexible hardware signature module that fits the requirements of the many tools and applications that can be potentially executed in a modern general purpose multicore processor. This chapter summarizes the main contributions of this thesis, exposes the conclusions and proposes future research lines.

We have improved a signature based HTM system by adding a filter mechanism that allows a reduction in the use of signatures in the baseline system. This filter (called CFM-TM) is implemented by slightly changing the cache coherence protocol and adding one bit to the L1 cache lines (it can maintain two versions of data in private L1 and shared L2 cache). CFM-TM allows us to reduce the write signature up to 75% (maintaining the false positive rate) while keeping approximately the same performance of the baseline system running alone (the CFM-TM can also be deactivated if needed), and it can improve the performance under certain conditions (as we prove with one of the benchmarks, which reduces the execution time more than 40%). To fully take advantage of this filter, the CMP should provide a flexible management of signatures.

We have proposed the first Hardware Asymmetric Data Race Detector.[1] Despite the fact that some software approaches have been developed before, our approach (called Pacman) is the first hardware approach that besides detecting asymmetric data races, is also able to tolerate them. Pacman requires minimum hardware support: it is implemented as a centralized module that hosts some signatures and some logic and that snoops the cache coherence

[1]This work was developed at the University of Illinois at Urbana-Champaign in collaboration with the members of the I-ACOMA group.

protocol traffic to detect and tolerate these races. Pacman temporally protects the variables accessed in critical sections in safe threads against buggy accesses from unsafe threads, and furthermore, it produces a negligible performance slowdown (the average execution time overhead is always under 0.1% in our evaluation). Our test shows the efficiency of Pacman, which was able to detect two unreported asymmetric data races in two well tested benchmarks, and to detect all the asymmetric races of a benchmark which was modified to artificially generate asymmetric data races.

We have also contributed with a flexible signature module to optimize the use of signatures in a general purpose multicore processor. From the two previous works, and from other tools for multicore processors, we realize that hardware signatures are too rigid to be really useful in general purpose multicore processors. We proposed a module (called *FlexSig*) that allows us to assign a variable number of resources to a signature depending on the needs of the requester. The *FlexSig* module can host a high number of signatures when the demand of signatures is high, and when the demand is low, *FlexSig* can achieve a very low false positive rate in the allocated signatures. We also propose several alternative allocation algorithms for improving the performance of the module by considering the asymmetry patterns of the requester needs. With *FlexSig* we can achieve very important improvements in the false positive rate (up to 100% in many cases when the demand is low). In summary, we have achieved a signature module that allocates signatures according to the demand, allowing from simple patterns of allocation (all the signatures with the same size) to more complex algorithms that assign more or less resources depending on the needs of the requester, and achieving important false positive rate reductions compared with conventional signatures.

Describing the thesis contributions in a nutshell, the performed experiments show novelty solutions in the area of Transactional Memory (TM) and Data Race Detection, with hardware signatures as the common element. We also develop a new flexible hardware signature module, designed to be placed in a multicore processor and to serve different tools and applications that require it. Despite these conclusions, there are some interesting ways to continue the work begun in this thesis. The future work can be open in several lines, of which we want to highlight the following:

– The CFM-TM filter (that we test in a HTM system) could be adapted to be used in other applications in a multicore processor, as for instance, speculative lock elision, that also requires speculation in their normal operation. Other techniques, tools, parallel abstractions or other speculative techniques could be also be benefited by an adaptation of our CFM-TM filter.

– The Pacman asymmetric data race detector could also be expanded for detecting races in other architectures, as for instance GPUs.

– Pacman could be expanded to detect more types of bugs and data races, such as atomicity violations, and to use it with another synchronization mechanisms (such as TM).

– We only test the *FlexSig* module in a HTM system. However, as we claim, *FlexSig* is designed to work in general purpose processors, which implies that many tools and applications may use it. We propose for future work to test our module in other tools, such as data race detectors, deterministic replay systems, other parallel abstractions, etc.

– Outside the world of multicore processors, signatures are also widely used in network applications, such as packet classification, packet inspection, routing protocols, etc. To test *FlexSig* in these scenarios could also be a good test to prove the flexibility and adaptability of *FlexSig*.

To conclude, the thesis has reached its goals of exploring the optimization in different tools for multicore processors, and of designing a signature module that fits in a general purpose multicore processor with very good results. Furthermore, the future work has also potential to continue exploring the possibilities of signatures in tools for supporting parallel programming, and to try to bring signatures near to a common resource in general purpose multicore processors by making them more flexible and a generally used.

Resumo da Tese

Seguindo o regulamento dos estudios de terceiro ciclo da Universidade de Santiago de Compostela, aprobado na Xunta de Goberno do día 7 de abril de 2000 (DOG de 6 de marzo de 2001) e modificado pola Xunta de Goberno de 14 de novembro de 2000, o Consello de Goberno de 22 de novembro de 2003, de 18 de xullo de 2005 (artigos 30 a 45), de 11 de novembro de 2008 e de 14 de maio de 2009; e, concretamente, cumprindo coas especificacións indicadas no capítulo 4, artigo 30, apartado 3 de dito regulamento, amósase a continuación un resumo en galego da presente tese.

Os procesadores multinúcleo comezaron unha nova era na que a programación paralela se fixo fundamental para continuar co escalado do rendemento nas novas aplicacións que se executan nestes procesadores. Porén, a programación paralela é mais difícil de codificar, mais propensa a erros, e máis difícil de comprender que a programación secuencial tradicional . O proceso de depuración paralela é unha das tarefas máis complicadas e que require maiores esforzos por parte dos programadores.

Para afrontar estas dificultades engadidas, e para que a programación paralela sexa aceptada e masivamente usada por a maioría dos programadores, é necesario desenvolver novas ferramentas que faciliten esta tarefa. Entre estas ferramentas podemos salientar dúas vertentes: aquelas que avogan por novas linguaxes de programación ou abstraccións especificamente pensadas para a programación paralela, e ferramentas que fagan máis sinxela as tarefas de depuración dos programas paralelos.

A raíz da dificultade da programación e depuración paralela reside en que todos os núcleos dun procesador multinúcleo comparten memoria, e polo tanto os programadores precisan da sincronización como mecanismo de control para acceder aos datos compartidos, e así non obter resultados inesperados. A ferramenta de sincronización máis utilizada son os ferrollos (locks), que protexen variables compartidas que o programador considera que deben de ser accedidas en exclusión mutua. Con todo, os ferrollos son difíciles de programar e son moi propensos a erros cando son utilizados en programación paralela.

Unha nova abstracción que cobrou moita importancia nos últimos anos dentro do mundo académico e da industria, é a Memoria Transaccional (TM, Transactional Memory) [72]. TM é unha abstracción software para a programación paralela que se basea na definición de rexións atómicas (transaccións) que o programador usa para garantir a exclusión mutua dun anaco de código. As características proporcionadas por esta abstracción son garantidos polo sistema TM implementado. TM foi pensado para mellorar os ferrollos como mecanismo de sincronización de memoria [173]. As transaccións deben ser executadas atomicamente e en illamento con respecto ao resto, mais poden executarse concorrentemente de maneira especulativa (usando para elo sistemas de detección de conflitos e dúas versións dos datos). O sistema de TM é o encargado de realizar esta especulación, sendo transparente para o programador. TM evita moitos dos posibeis fallos xerados polos ferrollos, tales como interbloqueo, interbloqueo activo ou inversión da prioridade.

Un sistema TM pode ser implementado en hardware (HTM), en software (STM), ou nunha mestura de ámbolos dous (HyTM). Porén, os sistemas STM normalmente non poden manter un nivel de rendemento equiparábel aos ferrollos, debido á complexidade que implica estes sistemas de tipo especulativo, mais si permiten unha flexibilidade total nas políticas de resolución de conflitos e de manexo das versións dos datos, a diferencia dos sistemas HTM e HyTM. Os sistemas HTM están implementados enteiramente en hardware, e teñen un rendemento moi superior a STM, e en moitos casos tamén superior aos ferrollos. Porén, estes sistemas teñen unha flexibilidade moi limitada. Por último, os sistemas HyTM son un compromiso entre as dúas implementacións anteriores, con un hardware limitado para executar algunhas tarefas (por exemplo detección de conflitos), e con unha parte software que da certa flexibilidade ao sistema. Nesta tese interésannos os sistemas con algún tipo de soporte hardware (HTM e HyTM).

Ademais de novas abstraccións e linguaxes de programación, tamén é moi importante a creación de ferramentas para acelerar o proceso de depuración. Tense demostrado que en moitos casos o tempo investido na programación de programas paralelos destínase en grande parte á tarefa de depuración do código, debido á dificultade para atopar e corrixir estes erros. A técnica do "printf" pode resultar útil en erros triviais, mais na maioría dos erros de concorrencia, esta técnica é insuficiente. Exemplos destas ferramentas de apoio á depuración son os detectores de condicións de carreira de datos, sistemas de re-execución determinista, etc.

Os erros máis frecuentes en programación paralela sobre arquitecturas multinúcleo son debidos ás carreira de datos. Unha condición de carreira ocorre cando dous ou mais fíos de execución acceden á mesma variable sen a sincronización axeitada, podendo resultar nun comportamento non desexado. Estes erros son especialmente difíciles de depurar por que normalmente os seus efectos non se fan notar de maneira inmediata, e son complicados de localizar. Ao igual que pasa con TM, existen sistemas de detección de condicións de carreira que están implementados en software [151] [48] [49] [92] [172], e outros que usan hardware para mellorar o seu rendemento [113] [132].

Entre os diferentes tipos de condicións de carreira, hai unha que nos interesa especialmente nesta tese por ser un problema real moi frecuente [140], e por non ter recibido até agora moita atención por parte da comunidade investigadora. Esta é a condición de carreira asimétrica, que ocorre cando unha un fío accede a variables compartidas coa sincronización adecuada, mentres que outro fío accede a estas mesmas variables sen ningún tipo de sincronización. O escenario típico no que ocorren estes erros é cando nun programa hai un fío correctamente sincronizado, e outro fío (tipicamente externo, tal como librerías, etc.) non o suficientemente ben probado e que accede a variables compartidas sen a sincronización adecuada, o que desencadea nun funcionamento imprevisíbel. Todos os detectores de condicións de carreira asimétricas previos están implementados en software [140] [137], e é nesta tese cando se propón o primeiro sistema hardware para detectar e ademais tolerar condicións de carreira asimétricas.

Todas estas ferramentas desenvolvidas para facilitar a accesibilidade da programación paralela teñen en común que, para obter un rendemento razoábel, precisan de algún tipo de soporte hardware. Por iso non é estraño que este tipo de ferramentas veñan acompañadas de aceleración hardware para manter un rendemento aceptábel. Para mostra, salientar que os principais fabricantes de procesadores multinúcleo (Intel, AMD, IBM) contan con procesadores con soporte hardware para TM, ben en forma de prototipos ou como procesadores comerciais que xa están no mercado (como por exemplo o Intel Haswell ou o Blue Gene/Q).

Un dos elementos de soporte hardware máis prometedores son as firmas [20] hardware, que poden ser usadas en grande variedade de ferramentas con diferentes propósitos, e proporcionan un aumento de rendemento significativo comparado co seu custo. As firmas son elementos de tamaño delimitado que poden albergar unha representación probabilística dun número ilimitado de direccións de memoria, e ademais poden comprobar se unha determinada dirección de memoria foi previamente introducida na firma. Unha firma está composta por un elemento de almacenamento (rexistro) e mais unha ou mais funcións de hash que serven para codificar as direccións cando son introducidas no rexistro. Para comprobar se unha dirección foi previamente introducida na firma, a dirección é codificada coas funcións de hash, que devolven as posicións dos bits do rexistro; se todos os bits están a un, a firma reporta que a dirección foi previamente introducida na firma, pero reporta que non foi introducida se cando menos un bit é cero. Este proceso pode reportar falsos positivos (reportar pertenza á firma cando en realidade non pertence) debido a que pode producirse solapamento entre as direccións da firma polo tamaño limitado de esta. Porén, nunca se van a producir falsos negativos (se unha dirección foi previamente introducida na firma, o resultado da comprobación será sempre positiva).

Contribucións

No contexto presentado anteriormente, a nosa tese móvese a cabalo entre ferramentas de soporte á programación paralela e as firmas como soporte hardware en procesadores multinúcleo. O noso obxectivo é explorar e optimizar o uso das firmas hardware como apoio para mellorar o rendemento de ferramentas relacionadas coa programación paralela, así como xeneralizar e flexibilizar o uso das firmas en procesadores multinúcleo de propósito xeral. Especificamente, nesta tese propoñemos: un filtro para sistemas Hardware TM (HTM) baseados en firmas hardware que permite reducir o tamaño das mesmas, o primeiro sistema hardware que detecta e tolera condicións de carreira asimétricas, e un módulo hardware de firmas flexibeis adecuado para satisfacer adecuadamente as demandas das diferentes aplicacións e ferramentas que o requiran nun procesador multinúcleo de propósito xeral. A continuación describimos estas contribucións en máis detalle.

Reducindo o Uso de Firmas nun Sistema HTM

Tomando como base un HTM baseado en firmas ven coñecido [183], deseñamos un filtro que permite reducir o uso das firmas hardware por parte do sistema base.

Este filtro, chamado CFM-TM [123], introduce lixeiras modificacións no protocolo de coherencia cache, así como un bit adicional (chamado *WTx*) en cada liña de memoria caché privada de primeiro nivel. Este bit adicional é o que indica (se o seu valor é un) que a liña cache está sendo xestionada polo CFM-TM, e polo tanto non ten que ser xestionado polo sistema HTM de base. A xestión por parte do filtro das liñas transaccionais é o seguinte: cando se produce un acceso de escritura dentro dunha transacción que pode ser manexado polo CFM-TM (como veremos, non todos os datos transaccionais poden ser manexados polo filtro), o bit *WTx* da liña correspondente ponse a un; con esta operación conseguimos manter a nova versión da liña transaccional (especulativa) na cache privada de primeiro nivel (L1), mentres que a versión vella da liña mantense na caché compartida de segundo nivel (L2). Ademais, este bit tamén será consultado para a detección de conflitos causados polos accesos doutros núcleos (aproveitando as mensaxes xeradas polo protocolo de coherencia caché).

Cando unha liña transaccional é manexada polo CFM-TM, esta non deixará o filtro até que remate a transacción. Isto implica que todas as liñas na caché L1 co bit *WTx* a un non poderán ser desaloxadas da mesma até que remate a transacción. Para conseguir isto, é preciso modificar o algoritmo de substitución de liñas cache, para que as liñas co bit *WTx* a un non poidan ser substituídas. Por esta razón, hai que poñer límites á capacidade do filtro para que a cache non chegue a un estado de interbloqueo.

Na avaliación do noso filtro observamos que se consegue unha redución da firma de escritura de até un 75%, conservando a mesma porcentaxe de falsos positivos que nun sistema sen filtro. Ademáis, o rendemento do sistema permanece practicamente inalterado coa introdución do filtro, e baixo certas circunstancias, conséguese mesmo un aumento considerábel do rendemento (para un dos programas usados, mellórase o rendemento do sistema global nun 40%).

Tolerando Condicións de Carreira Asimétricas con un Módulo de Firmas Hardware

Neste traballo deseñamos o primeiro módulo de firmas hardware (chamado Pacman) que detecta e tolera condicións de carreira asimétricas [132] [1] (todas as propostas anteriores foran desenvolvidas en software). Este módulo aproveita a información contida nas mensaxes do protocolo de coherencia caché para facer o seguimento das liñas caché accedidas durante as seccións críticas, así como aqueles accesos fora das rexións críticas coas que poden ter conflito. Mediante unhas mínimas modificacións no protocolo de coherencia, Pacman pode pro-

[1] Este traballo foi desenvolvido na Universidade de Illinois en Urbana-Champaign en colaboración cos membros do grupo I-ACOMA.

texer e tolerar condicións de carreira asimétricas. A información de todas as liñas é gardada por Pacman en firmas hardware, unha por cada posíbel fío de execución.

O funcionamento básico de Pacman é o seguinte: cando un fío de execución entra nunha sección crítica, Pacman activa a firma asociada a ese fío de execución, e empeza a introducir todas as direccións das liñas caché ás que se está accedendo nesa sección crítica; no caso de que outro fío de execución intente acceder de maneira non sincronizada a algunha dirección previamente introducida na firma, a condición de carreira asimétrica é detectada. Ademais, Pacman permite que o fío correctamente sincronizado continúe a súa execución, mentres que o fío que accede de maneira non sincronizada é paralizado por Pacman até que o outro fío sae da súa rexión crítica.

Pacman sitúase na rede de interconexión do procesador multinúcleo, e é capaz de rastrear todos os mensaxes de coherencia do sistema. Porén, hai situacións nas que o protocolo de coherencia caché non xera ningunha mensaxe, pero que Pacman debería rexistrar para o seu correcto funcionamento. Hai tres tipos de situacións: (1) cando un fío entra nunha sección crítica, algunhas liñas caché poden estar nun estado que permita ao fío acceder a estas liñas sen xerar ningunha mensaxe de coherencia, (2) cando se producen substitucións silenciosas de liñas caché (que non xeran mensaxes de coherencia) durante as seccións críticas e (3), e nas operacións de sincronización (adquirir e liberar os ferrollos). Todas estas situacións resólvense incluíndo modificacións moi pequenas no protocolo de coherencia para xerar mensaxes adicionais que apenas teñen influencia no rendemento do sistema. Ademais, Pacman resolve eficientemente algunhas situacións de potenciais bloqueos do sistema.

A implementación hardware de Pacman é moi sinxela, ten unha repercusión case nula no tempo de execución dos programas (nas probas realizadas, a media de deterioro do rendemento non supera o 0.1%), e consegue detectar con eficiencia as condicións de carreira asimétricas. Concretamente, nas nosas probas Pacman foi capaz de detectar dous erros que non foran descubertos antes en dous programas robustos, e todas as condicións de carreira asimétricas dun programa que foi modificado artificialmente para xerar condicións de carreira asimétricas de maneira continua.

Implementando un Módulo Flexíbel de Firmas Hardware

Os traballos con firmas realizados con TM e con detección de condicións de carreira, xunto con moitos traballos actuais para facilitar e mellorar a programación paralela, servíronos para darnos conta de que as firmas son unhas estruturas hardware que poden ser usadas en moitas ferramentas, polo que teñen potencial para ser incorporadas en procesadores multinúcleo de propósito xeral. Porén, as firmas hardware teñen unha estrutura demasiado ríxida para que poida ser usada de maneira xeral.

Neste traballo propoñemos un módulo flexíbel de firmas hardware para optimizar o uso das firmas nun procesador multinúcleo de propósito xeral. Este módulo, chamado *FlexSig* [122], asigna firmas hardware de tamaño variable (en función da demanda) ás ferramentas e aplicacións que as solicitan. *FlexSig* pode albergar unha grande cantidade de firmas cando a súa demanda é moi alta, e cando a demanda e baixa, *FlexSig* consegue unha taxa de falsos positivos moi baixa.

O funcionamento de *FlexSig* basease nos seguintes principios:

– *FlexSig* está composto por un certo número de recursos que poden ser agrupados para formar firmas de diferentes tamaños.

– *FlexSig* sempre trata de usar o máximo número de recursos e repartilos equitativamente entre as firmas presentes no módulo.

– As firmas aloxadas en *FlexSig* poden decrecer en tamaño en tempo de execución (isto repercute no número de falsos positivos), pero non poden aumentar.

Cando se queren asignar recursos de *FlexSig* para albergar unha nova firma, pódense dar tres casos, que *FlexSig* resolve da seguinte maneira:

1. *FlexSig* ten todos os recursos libres: todos os recursos de *FlexSig* son asignados á nova firma.

2. *FlexSig* ten todos os seus recursos ocupados: libéranse algúns recursos das firmas que teñen recursos xa asignados en *FlexSig* para deixar espazo para a nova firma, sempre tendo en conta o principio de equidade entre firmas.

3. *FlexSig* ten parte dos seus recursos sen usar, e parte deles asignados a firmas: asígnanse á nova firma os recursos libres da *FlexSig*, e se non son suficientes para cumprir o principio de equidade entre firmas, libéranse os recursos necesarios de outras firmas para manter este principio.

A avaliación de *FlexSig* está feita nun sistema de TM, e mostra a grande redución no número de falsos positivos de *FlexSig* con respecto a firmas convencionais. Especificamente, *FlexSig* mellora considerablemente o rendemento das firmas convencionais cando o número de transaccións medias no módulo é baixo, e ao mesmo tempo, é capaz de manexar moitas transaccións ao mesmo tempo (a costa dos falsos positivos), o que lle confire unha gran flexibilidade.

Asignación Asimétrica nun Módulo Flexíbel de Firmas para Procesadores Multinúcleo

FlexSig ten un algoritmo de asignación de recursos simétrico, é dicir, asigna os recursos de forma igualitaria entre todos os solicitantes. Porén, moitas das aplicacións e ferramentas que usan firmas non requiren os mesmos tamaños de firma, polo que o comportamento de *FlexSig* ten moito potencial de mellora en aplicacións deste tipo.

Esta proposta expón diferentes algoritmos asimétricos de varios niveis baseados en prioridades para a asignación de recursos ás firmas. Na avaliación usamos un sistema TM con unha firma de lectura e outra de escritura por transacción. Isto permítenos distinguir dous niveis de prioridade: un referente aos distintos tamaños das transaccións (as transaccións mais grandes ou que requiren mais recursos terán unha prioridade alta, e os que necesitan menos terán unha prioridade baixa), e outro, interno a cada transacción, que ten en conta o tamaño relativo do conxunto de lectura e do conxunto de escritura (normalmente a firma de lectura terá maior prioridade que a de escritura).

Neste traballo describimos os algoritmos en detalle, e ademais propoñemos unha implementación hardware altamente eficiente e paralela, tanto para algoritmos simétricos como asimétricos.

A nosa avaliación experimental explora diferentes combinacións dos dous niveis de prioridades expostos anteriormente. Os resultados evidencian unha mellora considerábel dos algoritmos asimétricos con respecto aos simétricos, conseguindo melloras superiores ao 60% en varios casos.

Conclusións e Traballo Futuro

Para concluír, en poucas palabras, podemos resumir as contribucións desta tese da seguinte maneira: elaboramos técnicas para a redución do uso de firmas nun sistema HTM, propoñemos a primeira aproximación hardware para detectar e tolerar condicións de carreira asimétricas baseada nun módulo de firmas, deseñamos un novo módulo de firmas flexíbel para ser usado por aplicacións e ferramentas en procesadores multinúcleo, e implementamos novos algoritmos para estes módulo de firmas flexíbeis que melloran de maneira considerábel os resultados dos anteriores.

Por último, o traballo futuro relacionado con esta tese podería ir dirixido en varios camiños, dos cales queremos destacar as seguintes:

- O filtro CFM-TM podería ser adaptado para o seu uso noutras aplicacións en procesadores multinúcleo, como por exemplo elisión especulativa de ferrollos, que comparte moitas características comúns con TM.
- Pacman podería ser adaptado para detectar condicións de carreira asimétricas noutras arquitecturas, como por exemplo GPUs.
- Pacman podería ser ampliado para detectar outro tipo de erros e condicións de carreira, como por exemplo violacións de atomicidade, e podería ser utilizado con outro tipo de mecanismos de sincronización (como TM).
- Dado que *FlexSig* foi probado soamente nun sistema TM, propoñemos probalo tamén con outras ferramentas como detectores de condicións de carreira, sistemas de reexecución determinista, etc.
- Fora dos procesadores multinúcleo, as firmas son tamén amplamente usadas en aplicacións de rede tales como clasificación de paquetes, inspección de paquetes, protocolos de encamiñamento, etc. Probar *FlexSig* neste escenario sería un bo test para a flexibilidade e adaptabilidade do módulo.

Bibliography

[1] Martín Abadi, Tim Harris, and Mojtaba Mehrara. Transactional memory with strong atomicity using off-the-shelf memory protection hardware. In *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '09, pages 185–196, New York, NY, USA, 2009. ACM.

[2] Ali-Reza Adl-Tabatabai, Brian T. Lewis, Vijay Menon, Brian R. Murphy, Bratin Saha, and Tatiana Shpeisman. Compiler and runtime support for efficient software transactional memory. In *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '06, pages 26–37, New York, NY, USA, 2006. ACM.

[3] M. Ahmadi and S. Wong. K-stage pipelined bloom filter for packet classification. In *Computational Science and Engineering, 2009. CSE '09. International Conference on*, volume 2, pages 64–70, 2009.

[4] P Almeida, C Baquero, N Preguica, and D Hutchison. Scalable Bloom Filters. *Information Processing Letters*, 101(6):255–261, March 2007.

[5] AMD. The industry-changing impact of accelerated computing, 2008.

[6] AMD. Advanced synchronization facility – proposed architectural specification, 2009.

[7] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM.

[8] C. Scott Ananian, Krste Asanovic, Bradley C. Kuszmaul, Charles E. Leiserson, and Sean Lie. Unbounded transactional memory. In *Proceedings of the 11th International*

Symposium on High-Performance Computer Architecture, pages 316–327, Washington, DC, USA, 2005. IEEE Computer Society.

[9] James P. Anderson, Samuel A. Hoffman, Joseph Shifman, and Robert J. Williams. D825 - a multiple-computer system for command & control. In *Proceedings of the December 4-6, 1962, fall joint computer conference*, AFIPS '62 (Fall), pages 86–96, New York, NY, USA, 1962. ACM.

[10] David F. Bacon and Seth Copen Goldstein. Hardware-assisted replay of multiprocessor programs. In *Proceedings of the 1991 ACM/ONR workshop on Parallel and distributed debugging*, PADD '91, pages 194–206, New York, NY, USA, 1991. ACM.

[11] David A. Bader and Kamesh Madduri. Design and implementation of the hpcs graph analysis benchmark on symmetric multiprocessors. In *Proceedings of the 12th international conference on High Performance Computing*, HiPC'05, pages 465–476, Berlin, Heidelberg, 2005. Springer-Verlag.

[12] D. H. Bailey. Ffts in external of hierarchical memory. In *Proceedings of the 1989 ACM/IEEE conference on Supercomputing*, Supercomputing '89, pages 234–242, New York, NY, USA, 1989. ACM.

[13] A. O. Balan, L. Sigal, and M. J. Black. A quantitative evaluation of video-based 3d person tracking. In *Proceedings of the 14th International Conference on Computer Communications and Networks*, ICCCN '05, pages 349–356, Washington, DC, USA, 2005. IEEE Computer Society.

[14] Prithviraj Banerjee. *Parallel algorithms for VLSI computer-aided design*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1994.

[15] Lee Baugh, Naveen Neelakantam, and Craig Zilles. Using hardware memory protection to build a high-performance, strongly-atomic hybrid transactional memory. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, pages 115–126, Washington, DC, USA, 2008. IEEE Computer Society.

[16] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: characterization and architectural implications. In *Proceedings of*

the 17th international conference on Parallel architectures and compilation techniques, PACT ’08, pages 72–81, New York, NY, USA, 2008. ACM.

[17] Fischer Black and Myron Scholes. The pricing of options and corporate liabilities. *Journal of Political Economy*, 81(3):pp. 637–654, 1973.

[18] G. Blake, R.G. Dreslinski, and T. Mudge. A survey of multicore processors. *Signal Processing Magazine, IEEE*, 26(6):26 –37, november 2009.

[19] Guy E. Blelloch, Charles E. Leiserson, Bruce M. Maggs, C. Greg Plaxton, Stephen J. Smith, and Marco Zagha. A comparison of sorting algorithms for the connection machine cm-2. In *Proceedings of the third annual ACM symposium on Parallel algorithms and architectures*, SPAA ’91, pages 3–16, New York, NY, USA, 1991. ACM.

[20] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, July 1970.

[21] U. Bondhugula, M. Baskaran, A. Hartono, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. Towards effective automatic parallelization for multicore systems. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1 –5, april 2008.

[22] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership types for safe programming: preventing data races and deadlocks. In *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA ’02, pages 211–230, New York, NY, USA, 2002. ACM.

[23] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for gpus: stream computing on graphics hardware. In *ACM SIGGRAPH 2004 Papers*, SIGGRAPH ’04, pages 777–786, New York, NY, USA, 2004. ACM.

[24] David Callahan and Ken Kennedy. Compiling programs for distributed-memory multiprocessors. *The Journal of Supercomputing*, 2:151–169, 1988.

[25] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IISWC ’08: Proceedings*

of The IEEE International Symposium on Workload Characterization, September 2008.

[26] J. Lawrence Carter and Mark N. Wegman. Universal classes of hash functions (extended abstract). In *Proceedings of the ninth annual ACM symposium on Theory of computing*, STOC ’77, pages 106–112, New York, NY, USA, 1977. ACM.

[27] Calin Cascaval, Colin Blundell, Maged Michael, Harold W. Cain, Peng Wu, Stefanie Chiras, and Siddhartha Chatterjee. Software transactional memory: Why is it only a research toy? *Queue*, 6(5):46–58, September 2008.

[28] Jared Casper, Tayo Oguntebi, Sungpack Hong, Nathan G. Bronson, Christos Kozyrakis, and Kunle Olukotun. Hardware acceleration of transactional memory on commodity systems. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, ASPLOS ’11, pages 27–38, New York, NY, USA, 2011. ACM.

[29] Luis Ceze, James Tuck, Pablo Montesinos, and Josep Torrellas. Bulksc: bulk enforcement of sequential consistency. In *Proceedings of the 34th annual international symposium on Computer architecture*, ISCA ’07, pages 278–289, New York, NY, USA, 2007. ACM.

[30] K. M. Chandy and J. Misra. The drinking philosophers problem. *ACM Trans. Program. Lang. Syst.*, 6(4):632–646, October 1984.

[31] Francis Chang, Kang Li, and Wu chang Feng. Approximate caches for packet classification. In *INFOCOM*, 2004.

[32] Shailender Chaudhry, Robert Cypher, Magnus Ekman, Martin Karlsson, Anders Landin, Sherman Yip, Håkan Zeffer, and Marc Tremblay. Rock: A high-performance sparc cmt processor. *IEEE Micro*, 29(2):6–16, March 2009.

[33] Jong-Deok Choi et al. Efficient and precise datarace detection for multithreaded object-oriented programs. In *PLDI*, 2002.

[34] Jong-Deok Choi and Harini Srinivasan. Deterministic replay of java multithreaded applications. In *Proceedings of the SIGMETRICS symposium on Parallel and distributed tools*, SPDT ’98, pages 48–59, New York, NY, USA, 1998. ACM.

[35] W. Choi and J. Draper. Improving utilization of hardware signatures in transactional memory. *Parallel and Distributed Systems, IEEE Transactions on*, PP(99):1–1, 2012.

[36] Woojin Choi and Jeff Draper. Unified signatures for improving performance in transactional memory. In *IPDPS*, pages 817–827. IEEE, 2011.

[37] Dave Christie, Jae-Woong Chung, Stephan Diestelhorst, Michael Hohmuth, Martin Pohlack, Christof Fetzer, Martin Nowack, Torvald Riegel, Pascal Felber, Patrick Marlier, and Etienne Rivière. Evaluation of amd's advanced synchronization facility within a complete transactional memory stack. In *Proceedings of the 5th European conference on Computer systems*, EuroSys '10, pages 27–40, New York, NY, USA, 2010. ACM.

[38] Jaewoong Chung, Luke Yen, Stephan Diestelhorst, Martin Pohlack, Michael Hohmuth, David Christie, and Dan Grossman. Asf: Amd64 extension for lock-free data structures and transactional memory. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 39–50, Washington, DC, USA, 2010. IEEE Computer Society.

[39] Cliff Click. And now some hardware transactional memory comments..., 2009.

[40] Cuda. `http://www.nvidia.com/object/cuda_home_new.html`.

[41] David Culler, J.P. Singh, and Anoop Gupta. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann, 1st edition, 1998. The Morgan Kaufmann Series in Computer Architecture and Design.

[42] Luke Dalessandro, Michael F. Spear, and Michael L. Scott. Norec: streamlining stm by abolishing ownership records. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 67–78, New York, NY, USA, 2010. ACM.

[43] Jonathan Deutscher and Ian Reid. Articulated body motion capture by stochastic search. *Int. J. Comput. Vision*, 61(2):185–205, February 2005.

[44] Joseph Devietti, Jacob Nelson, Tom Bergan, Luis Ceze, and Dan Grossman. Rcdc: a relaxed consistency deterministic computer. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, ASPLOS XVI, pages 67–78, New York, NY, USA, 2011. ACM.

[45] Dave Dice, Yossi Lev, Mark Moir, and Daniel Nussbaum. Early experience with a commercial hardware transactional memory implementation. In *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems*, ASPLOS '09, pages 157–168, New York, NY, USA, 2009. ACM.

[46] Dave Dice, Ori Shalev, and Nir Shavit. Transactional locking ii. In *Proceedings of the 20th international conference on Distributed Computing*, DISC'06, pages 194–208, Berlin, Heidelberg, 2006. Springer-Verlag.

[47] Edsger W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Inf.*, 1:115–138, 1971.

[48] Dawson Engler and Ken Ashcraft. Racerx: effective, static detection of race conditions and deadlocks. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, pages 237–252, New York, NY, USA, 2003. ACM.

[49] John Erickson, Madanlal Musuvathi, Sebastian Burckhardt, and Kirk Olynyk. Effective data-race detection for the kernel. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–16, Berkeley, CA, USA, 2010. USENIX Association.

[50] Apache Software Foundation. *Apache HTTP Server Reference Manual - for Apache version 2.2.17*. Network Theory Ltd., 2010.

[51] Keir Fraser and Tim Harris. Concurrent programming without locks. *ACM Trans. Comput. Syst.*, 25(2), May 2007.

[52] Transactional memory in gcc. `http://gcc.gnu.org/wiki/TransactionalMemory`.

[53] D. Geer. Chip makers turn to multicore processors. *Computer*, 38(5):11 – 13, may 2005.

[54] Kourosh Gharachorloo. Memory consistency models for shared-memory multiprocessors. Technical report, 1995.

[55] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th annual international symposium on Computer Architecture*, ISCA '90, pages 15–26, New York, NY, USA, 1990. ACM.

[56] Anwar Ghuloum. What makes parallel programming hard?, 2007.

[57] P.B. Gibbons. What good are shared-memory models? In *Parallel Processing, 1996. Proceedings of the 1996 ICPP Workshop on Challenges for*, pages 103 –114, aug 1996.

[58] Leslie Frederick Greengard. *The rapid evaluation of potential fields in particle systems*. PhD thesis, New Haven, CT, USA, 1987. AAI8727216.

[59] M. Gries, U. Hoffmann, M. Konow, and M. Riepen. Scc: A flexible architecture for many-core platform research. *Computing in Science Engineering*, 13(6):79 –83, nov.-dec. 2011.

[60] Rachid Guerraoui, Maurice Herlihy, and Bastian Pochon. *Polymorphic Contention Management*. 2005.

[61] Deke Guo, Jie Wu, Honghui Chen, Ye Yuan, and Xueshan Luo. The dynamic bloom filters. *IEEE Trans. on Knowl. and Data Eng.*, 22(1):120–133, January 2010.

[62] Lance Hammond, Peter G. Gyarmati, Christos Kozyrakis, Kunle Olukotun, Brian D. Carlstrom, Ben Hertzberg, Vicky Wong, Mike Chen, John D. Davis, Manohar K. Prabhu, and Honggo Wijaya. Transactional memory coherence and consistency (tcc). In *IN PROCEEDINGS OF THE 11TH INTL. SYMPOSIUM ON COMPUTER ARCHITECTURE (ISCA*, 2004.

[63] Lance Hammond, Mark Willey, and Kunle Olukotun. Data speculation support for a chip multiprocessor. In *Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, ASPLOS-VIII, pages 58–69, New York, NY, USA, 1998. ACM.

[64] Pat Hanrahan, David Salzman, and Larry Aupperle. A rapid hierarchical radiosity algorithm. In *Proceedings of the 18th annual conference on Computer graphics and*

interactive techniques, SIGGRAPH ’91, pages 197–206, New York, NY, USA, 1991. ACM.

[65] P.B. Hansen. The programming language concurrent pascal. *Software Engineering, IEEE Transactions on*, SE-1(2):199 –207, june 1975.

[66] Ruud Haring, Martin Ohmacht, Thomas Fox, Michael Gschwind, David Satterfield, Krishnan Sugavanam, Paul Coteus, Philip Heidelberger, Matthias Blumrich, Robert Wisniewski, alan gara, George Chiu, Peter Boyle, Norman Chist, and Changhoan Kim. The ibm blue gene/q compute chip. *Micro, IEEE*, 32(2):48 –60, march-april 2012.

[67] T. Harris, A. Cristal, O.S. Unsal, E. Ayguade, F. Gagliardi, B. Smith, and M. Valero. Transactional memory: An overview. *Micro, IEEE*, 27(3):8 –29, may-june 2007.

[68] Tim Harris, Mark Plesko, Avraham Shinnar, and David Tarditi. Optimizing memory transactions. In *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, PLDI ’06, pages 14–25, New York, NY, USA, 2006. ACM.

[69] David Heath, Robert Jarrow, and Andrew Morton. Bond pricing and the term structure of interest rates: A new methodology for contingent claims valuation. *Econometrica*, 60(1):77–105, January 1992.

[70] Mark Andrew Heinrich. *The performance and scalability of distributed shared-memory cache coherence protocols*. PhD thesis, Stanford, CA, USA, 1999. AAI9924431.

[71] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the twenty-second annual symposium on Principles of distributed computing*, PODC ’03, pages 92–101, New York, NY, USA, 2003. ACM.

[72] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. In *Proceedings of the 20th annual international symposium on computer architecture*, ISCA ’93, pages 289–300, New York, NY, USA, 1993. ACM.

[73] Sungpack Hong, T. Oguntebi, J. Casper, N. Bronson, C. Kozyrakis, and K. Olukotun. Eigenbench: A simple exploration tool for orthogonal tm characteristics. In *Workload Characterization (IISWC), 2010 IEEE International Symposium on*, pages 1 –11, dec. 2010.

[74] Version Sparc International, David L. Weaver, and Tom Germond. The sparc architecture manual, 1992.

[75] Introduction to parallel computing. `https://computing.llnl.gov/tutorials/parallel_comp/`.

[76] S. Burckhardt J. Erickson, M. Musuvathi and K. Olynyk. Effective data-race detection for the kernel, October 2010.

[77] Christian Jacobi, Timothy Slegel, and Dan Greiner. Transactional memory architecture and implementation for ibm system z. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '12, pages 25–36, Washington, DC, USA, 2012. IEEE Computer Society.

[78] Chi Jing. Application and research on weighted bloom filter and bloom filter in web cache. In *Web Mining and Web-based Application, 2009. WMWA '09. Second Pacific-Asia Conference on*, pages 187–191, 2009.

[79] C. R. Johns and D. A. Brokenshire. Introduction to the cell broadband engine architecture. *IBM Journal of Research and Development*, 51(5):503 –519, sept. 2007.

[80] David Kanter. Analysis of haswell's transactional memory, http://www.realworldtech.com/page.cfm?articleid=rwt021512050738, 2012.

[81] Chi keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa, and Reddi Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *In Programming Language Design and Implementation*, pages 190–200. ACM Press, 2005.

[82] T. Kocak and I. Kaya. Low-power bloom filter architecture for deep packet inspection. *Communications Letters, IEEE*, 10(3):210–212, 2006.

[83] Kunal Korgaonkar, Kashyap Garimella, and Kamakoti Veezhinathan. Size-proportional signature sharing for transactional memory systems. *FASPP workshop*, June 2012.

[84] Sanjeev Kumar, Michael Chu, Christopher J. Hughes, Partha Kundu, and Anthony Nguyen. Hybrid transactional memory. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '06, pages 209–220, New York, NY, USA, 2006. ACM.

[85] R. M. N. Mishra L. O'Callaghan, A. Meyerson and S. Guha. High-performance clustering of streams and large data sets. In *In Proceedings of the 18th International Conference on Data Engineering*, February 2002.

[86] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, 28(9):690–691, September 1979.

[87] James R. Larus and Ravi Rajwar. *Transactional Memory*. Morgan & Claypool, 2006.

[88] Charles E. Leiserson. The cilk++ concurrency platform. In *Proceedings of the 46th Annual Design Automation Conference*, DAC '09, pages 522–527, New York, NY, USA, 2009. ACM.

[89] Xueming Li, Lijuan Peng, and Chunlin Zhang. Application of bloom filter in grid information service. In *Multimedia Information Networking and Security (MINES), 2010 International Conference on*, pages 866–870, 2010.

[90] Changhui Lin, Vijay Nagarajan, and Rajiv Gupta. Efficient sequential consistency using conditional fences. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, pages 295–306, New York, NY, USA, 2010. ACM.

[91] D. B. Lomet. Process structuring, synchronization, and recovery using atomic actions. In *Proceedings of an ACM conference on Language design for reliable software*, pages 128–137, New York, NY, USA, 1977. ACM.

[92] Shan Lu, Joseph Tucek, Feng Qin, and Yuanyuan Zhou. Avio: detecting atomicity violations via access interleaving invariants. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, ASPLOS-XII, pages 37–48, New York, NY, USA, 2006. ACM.

[93] Shan Lu, Joseph Tucek, Feng Qin, and Yuanyuan Zhou. Avio: Detecting atomicity violations via access interleaving invariants. In *In ASPLOS*, pages 37–48, 2006.

[94] Brandon Lucia, Joseph Devietti, Karin Strauss, and Luis Ceze. Atom-aid: Detecting and surviving atomicity violations. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, pages 277–288, Washington, DC, USA, 2008. IEEE Computer Society.

[95] Marc Lupon, Grigorios Magklis, and Antonio Gonzalez. Fastm: A log-based hardware transactional memory with fast abort recovery. In *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*, PACT '09, pages 293–302, Washington, DC, USA, 2009. IEEE Computer Society.

[96] Marc Lupon, Grigorios Magklis, and Antonio Gonzalez. A dynamically adaptable hardware transactional memory. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 27–38, Washington, DC, USA, 2010. IEEE Computer Society.

[97] Qin Lv, William Josephson, Zhe Wang, Moses Charikar, and Kai Li. Ferret: a toolkit for content-based similarity search of feature-rich data. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, EuroSys '06, pages 317–330, New York, NY, USA, 2006. ACM.

[98] Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hållberg, Johan Högberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. Simics: A full system simulation platform. *Computer*, 35(2):50–58, February 2002.

[99] Virendra J. Marathe, William N. Scherer, and Michael L. Scott. Adaptive software transactional memory. In *Proceedings of the 19th international conference on Distributed Computing*, DISC'05, pages 354–368, Berlin, Heidelberg, 2005. Springer-Verlag.

[100] Virendra J. Marathe, Michael F. Spear, Christopher Heriot, Athul Acharya, David Eisenstat, William N. Scherer Iii, and Michael L. Scott. Lowering the overhead of nonblocking software transactional memory. In *Dept. of Computer Science, Univ. of Rochester*, 2006.

[101] Milo Martin, Colin Blundell, and E. Lewis. Subtleties of transactional memory atomicity semantics. *IEEE Comput. Archit. Lett.*, 5(2):17–17, July 2006.

[102] Milo M. K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, and David A. Wood. Multifacet's general execution-driven multiprocessor simulator (gems) toolset. *SIGARCH Comput. Archit. News*, 33(4):92–99, November 2005.

[103] K Martinez and J Cupitt. Vips - a highly tuned image processing software architecture. In *IEEE International Conference on Image Processing*, volume 2, pages 574–577. IEEE, 2005. Event Dates: Sept. 2005.

[104] Mojtaba Mehrara, Jeff Hao, Po-Chun Hsu, and Scott Mahlke. Parallelizing sequential applications on commodity hardware using a low-cost software transactional memory. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 166–176, New York, NY, USA, 2009. ACM.

[105] Xstm. `http://www.xstm.net`.

[106] Chi Cao Minh, Martin Trautmann, JaeWoong Chung, Austen McDonald, Nathan Bronson, Jared Casper, Christos Kozyrakis, and Kunle Olukotun. An effective hybrid transactional memory system with strong isolation guarantees. In *Proceedings of the 34th annual international symposium on Computer architecture*, ISCA '07, pages 69–80, New York, NY, USA, 2007. ACM.

[107] Pablo Montesinos, Luis Ceze, and Josep Torrellas. Delorean: Recording and deterministically replaying shared-memory multiprocessor execution eficiently. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, pages 289–300, Washington, DC, USA, 2008. IEEE Computer Society.

[108] G. E. Moore. Cramming More Components onto Integrated Circuits. *Electronics*, 38(8):114–117, April 1965.

[109] Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill, and David A. Wood. Logtm: Log-based transactional memory. In *in HPCA*, pages 254–265, 2006.

[110] Message passing interface (mpi).
`https://computing.llnl.gov/tutorials/mpi/`.

[111] Matthias Müller, David Charypar, and Markus Gross. Particle-based fluid simulation for interactive applications. In *Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation*, SCA '03, pages 154–159, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association.

[112] A. Muzahid, N. Otsuki, and J. Torrellas. Atomtracker: A comprehensive approach to atomic region inference and violation detection. In *Microarchitecture (MICRO), 2010 43rd Annual IEEE/ACM International Symposium on*, pages 287 –297, dec. 2010.

[113] Abdullah Muzahid, Dario Suárez, Shanxiang Qi, and Josep Torrellas. Sigrace: signature-based data race detection. In *Proceedings of the 36th annual international symposium on Computer architecture*, ISCA '09, pages 337–348, New York, NY, USA, 2009. ACM.

[114] Satish Narayanasamy, Cristiano Pereira, and Brad Calder. Recording shared memory dependencies using strata. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, ASPLOS-XII, pages 229–240, New York, NY, USA, 2006. ACM.

[115] Satish Narayanasamy, Gilles Pokam, and Brad Calder. Bugnet: Continuously recording program execution for deterministic replay debugging. In *Proceedings of the 32nd annual international symposium on Computer Architecture*, ISCA '05, pages 284–295, Washington, DC, USA, 2005. IEEE Computer Society.

[116] R. Netzer and B. Miller. Improving the accuracy of data race detection. In *PPoPP*, 1991.

[117] Nvidia. The benefits of multiple cpu cores in mobile devices., 2010.

[118] M. Olszewski, J. Cutler, and J.G. Steffan. Judostm: A dynamic binary-rewriting approach to software transactional memory. In *Parallel Architecture and Compilation Techniques, 2007. PACT 2007. 16th International Conference on*, pages 365 –375, sept. 2007.

[119] Opencl - the open standard for parallel programming of heterogeneous systems. `http://www.khronos.org/opencl/`.

[120] Openmp. `http://openmp.org/wp/`.

[121] Dstm2 oracle corporation. `http://www.oracle.com/us/sun/index.html`.

[122] Lois Orosa, Elisardo Antelo, and Javier D. Bruguera. Flexsig: Implementing flexible hardware signatures. *ACM Trans. Archit. Code Optim.*, 8(4):30:1–30:20, January 2012.

[123] Lois Orosa, Javier D. Bruguera, and Elisardo Antelo. A cache filtering mechanism for hardware transactional memory systems decoupled from caches. In *XX Jornadas de Paralelismo*, A Coruña (Spain), September 2009.

[124] P. M. Ortego and P. Sack. SESC: SuperESCalar Simulator. December 2004.

[125] T. Osano, Y. Uchida, and N. Ishikawa. Routing protocol using bloom filters for mobile ad hoc networks. In *Mobile Ad-hoc and Sensor Networks, 2008. MSN 2008. The 4th International Conference on*, pages 89–94, 2008.

[126] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. In *Proceedings of the 9th Annual European Symposium on Algorithms*, ESA '01, pages 121–133, London, UK, UK, 2001. Springer-Verlag.

[127] S.K. Pal, P. Sardana, and K. Yadav. Efficient multilingual keyword search using bloom filter for cloud computing applications. In *Advanced Computing (ICoAC), 2012 Fourth International Conference on*, pages 1–7, 2012.

[128] Harish Patil, Cristiano Pereira, Mack Stallcup, Gregory Lueck, and James Cownie. Pinplay: a framework for deterministic replay and reproducible analysis of parallel programs. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, CGO '10, pages 2–11, New York, NY, USA, 2010. ACM.

[129] Lin Peng, Lun guo Xie, Xiao qiang Zhang, and Xin yan Xie. Conflict detection via adaptive signature for software transactional memory. In *Computer Engineering and Technology (ICCET), 2010 2nd International Conference on*, volume 2, pages V2–306 –V2–310, april 2010.

[130] Milos Prvulovic and Josep Torrellas. Reenact: Using thread-level speculation mechanisms to debug data races in multithreaded codes. In *International Symposium on Computer architecture*, 2003.

[131] Posix threads programming. `https://computing.llnl.gov/tutorials/pthreads/`.

[132] Shanxiang Qi, N. Otsuki, Lois Orosa, A. Muzahid, and J. Torrellas. Pacman: Tolerating asymmetric data races with unintrusive hardware. In *High Performance Computer Architecture (HPCA), 2012 IEEE 18th International Symposium on*, pages 1 –12, feb. 2012.

[133] Feng Qin, Joseph Tucek, Jagadeesan Sundaresan, and Yuanyuan Zhou. Rx: treating bugs as allergies—a safe method to survive software failures. In *Proceedings of the twentieth ACM symposium on Operating systems principles*, SOSP '05, pages 235–248, New York, NY, USA, 2005. ACM.

[134] Ricardo Quislant, Eladio Gutierrez, Oscar Plata, and Emilio L. Zapata. Improving signatures by locality exploitation for transactional memory. In *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*, pages 303–312, Washington, DC, USA, 2009. IEEE Computer Society.

[135] Ricardo Quislant, Eladio Gutierrez, Oscar Plata, and Emilio L. Zapata. Ls-sig: Locality-sensitive signatures for transactional memory. *IEEE Transactions on Computers*, 99(PrePrints), 2011.

[136] Ricardo Quislant, Eladio Gutierrez, Oscar Plata, and Emilio L. Zapata. Hardware signature designs to deal with asymmetry in transactional data sets. *IEEE Transactions on Parallel and Distributed Systems*, 99(PrePrints), 2012.

[137] Sriram Rajamani, G. Ramalingam, Venkatesh Prasad Ranganath, and Kapil Vaswani. Isolator: dynamically ensuring isolation in comcurrent programs. In *ASPLOS '09: Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, pages 181–192, New York, NY, USA, 2009. ACM.

[138] Ravi Rajwar and James R. Goodman. Speculative lock elision: enabling highly concurrent multithreaded execution. In *Proceedings of the 34th annual ACM/IEEE*

international symposium on Microarchitecture, MICRO 34, pages 294–305, Washington, DC, USA, 2001. IEEE Computer Society.

[139] M. V. Ramakrishna, E. Fu, and E. Bahcekapili. Efficient hardware hashing functions for high performance computers. *IEEE Trans. Comput.*, 46:1378–1381, December 1997.

[140] Paruj Ratanaworabhan, Martin Burtscher, Darko Kirovski, Benjamin Zorn, Rahul Nagpal, and Karthik Pattabiraman. Detecting and tolerating asymmetric races. In *PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 173–184, New York, NY, USA, 2009. ACM.

[141] James Reinders. *Intel threading building blocks - outfitting C++ for multi-core processor parallelism.* O'Reilly, 2007.

[142] Jose Renau, Basilio Fraguela, James Tuck, Wei Liu, Milos Prvulovic, Luis Ceze, Smruti Sarangi, Paul Sack, Karin Strauss, and Pablo Montesinos. SESC simulator, January 2005. http://sesc.sourceforge.net.

[143] Torvald Riegel, Christof Fetzer, and Pascal Felber. Time-based transactional memory with scalable time bases. In *Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, SPAA '07, pages 221–228, New York, NY, USA, 2007. ACM.

[144] Jim Ruppert. A delaunay refinement algorithm for quality 2-dimensional mesh generation. In *Selected papers from the fourth annual ACM SIAM symposium on Discrete algorithms*, SODA '93, pages 548–585, Orlando, FL, USA, 1995. Academic Press, Inc.

[145] Bratin Saha, Ali-Reza Adl-Tabatabai, Anwar Ghuloum, Mohan Rajagopalan, Richard L. Hudson, Leaf Petersen, Vijay Menon, Brian Murphy, Tatiana Shpeisman, Eric Sprangle, Anwar Rohillah, Doug Carmean, and Jesse Fang. Enabling scalability and performance in a large scale cmp environment. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 73–86, New York, NY, USA, 2007. ACM.

[146] Bratin Saha, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Chi Cao Minh, and Benjamin Hertzberg. Mcrt-stm: a high performance software transactional memory

system for a multi-core runtime. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '06, pages 187–197, New York, NY, USA, 2006. ACM.

[147] Daniel Sanchez, Luke Yen, Mark D. Hill, and Karthikeyan Sankaralingam. Implementing signatures for transactional memory. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, pages 123–133, Washington, DC, USA, 2007. IEEE Computer Society.

[148] Sutirtha Sanyal, Sourav Roy, Adrian Cristal, Osman S. Unsal, and Mateo Valero. Dynamically filtering thread-local variables in lazy-lazy hardware transactional memory. In *Proceedings of the 2009 11th IEEE International Conference on High Performance Computing and Communications*, pages 171–179, Washington, DC, USA, 2009. IEEE Computer Society.

[149] Vijay Saraswat, Bard Bloom, Igor Peshansky, Olivier Tardieu, and David Grove. X10 language specification. Technical report, IBM, January 2012.

[150] Ruchira Sasanka, Man-Lap Li, Sarita V. Adve, Yen-Kuang Chen, and Eric Debes. Alp: Efficient support for all levels of parallelism for complex media applications. *ACM Trans. Archit. Code Optim.*, 4(1), March 2007.

[151] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, November 1997.

[152] William N. Scherer and Michael L. Scott. Contention Management in Dynamic Software Transactional Memory, April 2004.

[153] Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, PODC '95, pages 204–213, New York, NY, USA, 1995. ACM.

[154] Zhang Shenghua, Qin Zheng, Zhao Yuan, and Peng Xiaolan. A cascade hash design of bloom filter for signature detection. In *Information Technology and Applications, 2009. IFITA '09. International Forum on*, volume 2, pages 559 –562, may 2009.

[155] Arrvindh Shriraman, Sandhya Dwarkadas, and Michael L. Scott. Flexible decoupled transactional memory support. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, pages 139–150, Washington, DC, USA, 2008. IEEE Computer Society.

[156] Eftychios Sifakis, Igor Neverov, and Ronald Fedkiw. Automatic determination of facial muscle activations from sparse motion capture marker data. In *ACM SIGGRAPH 2005 Papers*, SIGGRAPH '05, pages 417–425, New York, NY, USA, 2005. ACM.

[157] Gurindar S. Sohi, Scott E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *Proceedings of the 22nd annual international symposium on Computer architecture*, ISCA '95, pages 414–425, New York, NY, USA, 1995. ACM.

[158] Michael F. Spear, Virendra J. Marathe, William N. Scherer, and Michael L. Scott. Conflict detection and validation strategies for software transactional memory. In *Proceedings of the 20th international conference on Distributed Computing*, DISC'06, pages 179–193, Berlin, Heidelberg, 2006. Springer-Verlag.

[159] Michael F. Spear, Maged M. Michael, and Christoph von Praun. Ringstm: scalable transactions with a single atomic instruction. In *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, SPAA '08, pages 275–284, New York, NY, USA, 2008. ACM.

[160] J. Gregory Steffan and Todd C. Mowry. The potential for using thread-level data speculation to facilitate automatic parallelization. pages 2–13, 1998.

[161] P. Stenstrom. A survey of cache coherence schemes for multiprocessors. *Computer*, 23(6):12 –24, june 1990.

[162] Yu Sun and Wei Zhang. On-line trace based automatic parallelization of java programs on multicore platforms. In *Interaction between Compilers and Computer Architectures (INTERACT), 2011 15th Workshop on*, pages 35–43, 2011.

[163] S. Tarkoma, C.E. Rothenberg, and E. Lagerspetz. Theory and practice of bloom filters for distributed systems. *Communications Surveys Tutorials, IEEE*, 14(1):131–155, 2012.

[164] Gadi Taubenfeld. *Synchronization Algorithms and Concurrent Programming*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2006.

[165] Joseph Teran, Eftychios Sifakis, Geoffrey Irving, and Ronald Fedkiw. Robust quasistatic finite elements and flesh simulation. In *Proceedings of the 2005 ACM SIGGRAPH/Eurographics symposium on Computer animation*, SCA '05, pages 181–190, New York, NY, USA, 2005. ACM.

[166] Tilera. http://www.cs.cmu.edu/ scandal/nesl/info.html, 2012.

[167] M. Tremblay and S. Chaudhry. A third-generation 65nm 16-core 32-thread plus 32-scout-thread cmt sparcÂ® processor. In *Solid-State Circuits Conference, 2008. ISSCC 2008. Digest of Technical Papers. IEEE International*, pages 82 –83, feb. 2008.

[168] Marc Tremblay. Transactional memory for a modern microprocessor. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, PODC '07, pages 1–1, New York, NY, USA, 2007. ACM.

[169] James Tuck, Wonsun Ahn, Luis Ceze, and Josep Torrellas. Softsig: software-exposed hardware signatures for code analysis and optimization. In *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, ASPLOS XIII, pages 145–156, New York, NY, USA, 2008. ACM.

[170] Hans Vandierendonck and Koen De Bosschere. Xor-based hash functions. *IEEE Trans. Comput.*, 54(7):800–812, July 2005.

[171] Jack Veenstra and Robert J. Fowler. Mint: A front end for efficient simulation of shared-memory multiprocessors. pages 201–207, 1994.

[172] Kaushik Veeraraghavan, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. Detecting and surviving data races using complementary schedules. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 369–384, New York, NY, USA, 2011. ACM.

[173] Haris Volos, Andres Jaan Tack, Michael M. Swift, and Shan Lu. Applying transactional memory to concurrency bugs. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 211–222, New York, NY, USA, 2012. ACM.

[174] David W. Wall. Limits of instruction-level parallelism. In *Proceedings of the fourth international conference on Architectural support for programming languages and operating systems*, ASPLOS IV, pages 176–188, New York, NY, USA, 1991. ACM.

[175] Amy Wang, Matthew Gaudet, Peng Wu, José Nelson Amaral, Martin Ohmacht, Christopher Barton, Raul Silvera, and Maged Michael. Evaluation of blue gene/q hardware support for transactional memories. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, PACT '12, pages 127–136, New York, NY, USA, 2012. ACM.

[176] Ian Watson, Chris Kirkham, and Mikel Lujan. A study of a transactional parallel routing algorithm. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, PACT '07, pages 388–398, Washington, DC, USA, 2007. IEEE Computer Society.

[177] Turner Whitted. An improved illumination model for shaded display. *Commun. ACM*, 23(6):343–349, June 1980.

[178] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The splash-2 programs: characterization and methodological considerations. In *Proceedings of the 22nd annual international symposium on Computer architecture*, ISCA '95, pages 24–36, New York, NY, USA, 1995. ACM.

[179] Jingyue Wu, Heming Cui, and Junfeng Yang. Bypassing races in live applications with execution filters. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–13, Berkeley, CA, USA, 2010. USENIX Association.

[180] Bin Xiao and Yu Hua. Using parallel bloom filters for multiattribute representation on network services. *Parallel and Distributed Systems, IEEE Transactions on*, 21(1):20–32, 2010.

[181] Min Xu, Rastislav Bodik, and Mark D. Hill. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In *Proceedings of the 30th annual international symposium on Computer architecture*, ISCA '03, pages 122–135, New York, NY, USA, 2003. ACM.

[182] Luke Yen. *Signatures in transactional memory systems*. PhD thesis, Madison, WI, USA, 2009. AAI3367330.

[183] Luke Yen, Jayaram Bobba, Michael R. Marty, Kevin E. Moore, Haris Volos, Mark D. Hill, Michael M. Swift, and David A. Wood. Logtm-se: Decoupling hardware transactional memory from caches. In *In HPCA 13*, pages 261–272, 2007.

[184] Luke Yen, Stark C. Draper, and Mark D. Hill. Notary: Hardware techniques to enhance signatures. In *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 41, pages 234–245, Washington, DC, USA, 2008. IEEE Computer Society.

[185] Jie Yu and Satish Narayanasamy. Tolerating concurrency bugs using transactions as lifeguards. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 263–274, Washington, DC, USA, 2010. IEEE Computer Society.

[186] Wei Zhang, Chong Sun, and Shan Lu. Conmem: detecting severe concurrency bugs through an effect-oriented approach. In *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, ASPLOS XV, pages 179–192, New York, NY, USA, 2010. ACM.

[187] Pin Zhou, Feng Qin, Wei Liu, Yuanyuan Zhou, and Josep Torrellas. iwatcher: Efficient architectural support for software debugging. In *Proceedings of the 31st annual international symposium on Computer architecture*, ISCA '04, pages 224–, Washington, DC, USA, 2004. IEEE Computer Society.