UNIVERSIDAD
DE MÁLAGA

Departamento de Arquitectura de Computadores

TESIS DOCTORAL

Optimizing Signatures in Hardware Transactional Memory Systems

Ricardo Quislant del Barrio

Octubre de 2012

Dirigida por:
Óscar Plata,
Eladio Gutiérrez
Emilio L. Zapata

Dr. D. Óscar Plata González.
Catedrático del Departamento de Arquitectura de Computadores de la Universidad de Málaga.

Dr. D. Eladio D. Gutiérrez Carrasco.
Profesor Titular del Departamento de Arquitectura de Computadores de la Universidad de Málaga.

Dr. D. Emilio López Zapata.
Catedrático del Departamento de Arquitectura de Computadores de la Universidad de Málaga.

CERTIFICAN:

Que la memoria titulada “Optimizing Signatures in Hardware Transactional Memory Systems”, ha sido realizada por D. Ricardo Quislant del Barrio bajo nuestra dirección en el Departamento de Arquitectura de Computadores de la Universidad de Málaga y constituye la Tesis que presenta para optar al grado de Doctor en Ingeniería Informática.

Málaga, Octubre de 2012

Dr. D. Óscar Plata González.
Codirector de la tesis.

Dr. D. Eladio D. Gutiérrez Carrasco.
Codirector de la tesis.

Dr. D. Emilio López Zapata.
Codirector de la tesis.
Director del Departamento de Arquitectura de Computadores.

A mis padres y hermanos

Agradecimientos

Esta tesis es el producto de la colaboración directa e indirecta de multitud de personas a las que quisiera dirigir unas breves palabras de agradecimiento con estas líneas.

En primer lugar, quiero dar las gracias a mis directores, Óscar Plata, Eladio Gutiérrez, y Emilio L. Zapata, por acogerme en su grupo de investigación y por su sabio consejo y su inestimable aportación, sin la cual muchos de los capítulos de esta tesis quedarían incompletos. Me gustaría resaltar el magnífico clima que desde el primer día ha predominado en las múltiples reuniones de trabajo, en las que el trato de igual a igual y un gran sentido del humor han sentado las bases de muchas de las ideas plasmadas en esta tesis. He de dar las gracias al doctor Ezequiel Herruzo, de la Universidad de Córdoba, por su recomendación para que entrara a formar parte de este equipo.

Me gustaría extender este agradecimiento a todas las personas que integran el Departamento de Arquitectura de Computadores de la Universidad de Málaga, y en especial a los técnicos y a la secretaria, Carmen, cuyo trabajo ha sido indispensable para que esto llegara a buen fin.

Debo agradecer la financiación recibida por parte del Ministerio de Educación, Cultura y Deporte del Reino de España, a cargo de los proyectos de investigación CICYT TIN2006-01078 y CICYT TIN2010-16144, y de la beca FPI BES-2007-17318 asociada al proyecto TIN2006-01078 que he tenido el privilegio de disfrutar durante estos años.

I must thank the HiPEAC European Network of Excellence, under the 7th framework programme IST-217068, whose funds made me enjoy a three-month investigation stay at Chalmers University of Technology, with the group of my host, Professor Per Stenström, formed by Mafijul Islam, Mridha Waliullah and Anurag Negi, among others. Thank you for your kind support.

I would also like to thank Dr. Luke Yen from the University of Wisconsin, Madison, for providing us with his patches to adapt the STAMP workloads to the GEMS simulator.

Por último, quisiera agradecer encarecidamente el soporte anímico y afectivo que me ha brindado toda la gente de mi entorno. A mis compañeros de laboratorio que han creado un ambiente de trabajo inigualable, Marina Martínez, Juan Lucena, Sergio Varona, Antonio Ruíz, Rosa Castillo, Adrián Tineo, Pilar Poyato, Lidia Fernández, Iván Romero, Miguel Ángel Sánchez, Maxi García, Alfredo Martínez, Fernando Barranco, Victoria y Manolo Martín, Javier Ríos, Juan Villalba, Juan Luna, Pau Corral, Francisco Jaime, Alberto Sanz, Antonio J. Dios, Miguel Ángel González, Antonio Muñoz, Manolo R. Cervilla, Manolo Pedrero, Carlos García, Sergio Muñoz y a todos aquellos que haya podido olvidar. A mis amigos en Córdoba, que son enormes y han supuesto una ineludible válvula de escape para el estrés. A la música. A Patricia, que ha sido un apoyo incondicional y ha aguantado todos mis altibajos. Y a mis padres y hermanos, muchas gracias.

Abstract

With the advent of chip multiprocessors, hardware manufactures have put a big burden on software development community. The majority of programmers are used to sequential programming, whereas writing high-performance multithreaded programs is currently mastered by a small group. Parallel programming is a complex task that requires an understanding of new hardware concepts, algorithms, and programming tools.

Transactional Memory (TM) emerges as an alternative to the conventional multithreaded programming to ease the writing of concurrent programs. The programmer is provided with the transaction construct, which defines a group of computations that are executed atomically and in isolation. Transactional memory establishes an optimistic concurrency model where transactions run in parallel unless a conflict is detected. Conflict detection is performed transparently by the transactional system, and it is critical for performance. In case of hardware-implemented transactional memory, conflict detection is usually carried out by means of signatures, which keep track of the addresses that have been accessed by each transaction. Such signatures are implemented as hashing structures, Bloom filters specifically, that can yield false positives when checking for membership of an address. False positives can seriously degrade the performance of the system.

In this thesis, we propose various signature optimizations for hardware transactional memory, focusing on the reduction of the false positive rate mainly. First, an alternative to Bloom filters is devised, called *interval filter*, which tracks addresses read and written by transactions in form of contiguous memory location chunks, so-called intervals, motivated by the fact that applications frequently exhibit locality access patterns. In the line of exploiting locality of reference, we propose *locality-sensitive signatures* that defines new maps for the hash functions of Bloom filters in order to reduce the number of bits inserted into the filter for those addresses nearby located. As a result, false conflicts are significantly reduced for transactions that exhibit spatial locality. We also propose two signature schemes two tackle the problem of asymmetry in transactional data sets: a *multiset signature* and a *reconfigurable asymmetric signature*. Transactions frequently show an uneven cardinality of their sets of read and written addresses, while read and write filters are usually implemented with the same size each. Multiset signatures merge both filters in a single one, so that read and write false positive rates equalize each other. Multiset signatures are in turn optimized by exploiting locality and other properties of data access patterns. As regards reconfigurable asymmetric signatures, they can be dynamically configured at runtime to devote different amount of hardware either to the read set or to the write set of transactions. Finally, we conducted a *scalability study* to show the response of our proposals, compared to the common schemes, in the presence of contention, large transactions and different number of cores.

All experiments in this thesis have been performed by using state-of-the-art hardware simulators and benchmarks of the field of transactional memory. Specifically, we used Wisconsin GEMS, along with Simics for the transactional memory and chip multiprocessor simulators. We used the entire Stanford STAMP benchmark suite, which is specially designed for transactional memory research, together with EigenBench, a novel synthetic benchmark for studying orthogonal characteristics of TM systems. Finally, CACTI and Synopsys were also used for hardware area, power and time estimates.

Contents

Agradecimientos I

Abstract III

Contents V

List of Figures VII

List of Tables IX

1.- Introduction **1**

1.1. The Shift to Chip Multiprocessors 1

1.2. The Complexity of Parallel Programming 2

1.3. The Transactional Memory Abstraction 3

1.4. Thesis Motivation and Contributions 4

1.5. Thesis Structure 6

2.- Background and Related Work **9**

2.1. Transactional Memory Basics 9

2.1.1. Programmers' View 9

2.1.2. Nested Transactions 10

2.1.3. Weak and Strong Isolation 12

2.1.4. Data Versioning 13
2.1.5. Conflict Detection and Resolution 14
2.2. Overview of Transactional Memory Implementations 16
2.2.1. Software Transactional Memory 17
2.2.2. Hardware Transactional Memory 19
First HTM Approach 19
Main HTM Proposals 22
HTM in Commercial Off-The-Shelf Multiprocessors 25
2.3. Related Work on Signatures 27
2.3.1. The Bloom Filter 28
2.3.2. Hash Functions for Bloom Filters 31
2.3.3. The Birthday Paradox Problem: a motivation 33
2.3.4. Signature Implementations 33
2.3.5. Bloom Filter Variants 36

3.- Methodology 39
3.1. Simulation Framework 39
3.1.1. Simics Simulator 39
3.1.2. GEMS Toolset 41
3.1.3. Target System 42
3.1.4. TMtool: A Simple Functional TM System 44
3.2. Benchmark Suite 46
3.2.1. Benchmark Description and Characterization 46
3.2.2. Benchmark Adaptation to the Simulation TM System Environment 50

4.- Locality-Aware Interval Filters 53
4.1. The Interval Filter 54
4.2. Experimental Evaluation 58

5.- Locality-Sensitive Signatures 63

5.1. Locality Sensitive Signatures 64

5.1.1. Implementation 66

5.1.2. Features 68

5.1.3. False Positive Analysis 70

5.1.4. Locality in Benchmarks 72

5.2. Experimental Evaluation 73

5.2.1. Methodology 74

5.2.2. Exploring Radius and Delta 75

5.2.3. Interthread vs. Intrathread Locality: The Effect of Additional False Sharing 82

5.2.4. Locality-Sensitive PBX Hashing 86

5.2.5. Saving Hardware 88

6.- Dealing With Asymmetry In Data Sets 91

6.1. Multiset and Reconfigurable Asymmetric Signatures 92

6.1.1. Regular Multiset Signatures 92

6.1.2. Parallel Multiset Signatures 93

6.1.3. Parallel Multiset Shared Signatures 94

6.1.4. Reconfigurable Asymmetric Signatures 96

6.1.5. Hash Functions 97

6.2. False Positive Analysis 99

6.3. Experimental Evaluation 104

6.3.1. Methodology 104

6.3.2. Regular and Parallel Multiset Signatures Results 105

6.3.3. Reconfigurable Asymmetric Signatures Results 108

6.3.4. Parallel Multiset Shared Signatures Results 110

6.3.5. Enhancing Multiset Signatures with Locality-Sensitive Hashing 113

6.3.6. Hardware Implementation 117

7.- Scalability Analysis 121

7.1. The EigenBench Benchmark 122

7.1.1. Modifications to EigenBench 124

7.1.2. Orthogonal TM Characteristics 125

7.2. Experimental Evaluation 126

7.2.1. Contention Results 127

7.2.2. Transaction Length Results 129

7.2.3. Concurrency Results 131

7.2.4. The Working Set Effect 132

Conclusions and Future Work 135

Appendices 141

A.- Resumen en español 141

A.1. Filtro de intervalos 144

A.2. Firma sensible a la localidad 146

A.3. Firmas multiconjunto y asimétrica 149

A.4. Estudio de escalabilidad 152

A.5. Conclusiones y trabajo futuro 153

Bibliography 159

List of Figures

2.1. TM language extension constructs for programmers. 11

2.2. Examples of possible deadlock situations in TM systems. 15

2.3. Example of programming Herlihy and Moss' HTM system. Fragment of a producer/consumer benchmark [45]. 21

2.4. Design of a Bloom filter. A false positive scenario. 29

2.5. False positive probability of regular Bloom filters. $M = 1024$ and $k \in \{1, 2, 4, 8\}$. 30

2.6. Bit selection and XOR hashing function implementation. 32

2.7. Regular Bloom filter vs. Parallel Bloom filter. Design and implementation. 34

3.1. Simics+GEMS architecture: the memory module, Ruby, can be driven by either Simics or Opal's out-of-order processor module. . 40

3.2. Simulated CMP system organization. 43

3.3. LogTM-SE hardware add-ons. 44

4.1. Interval filter design. 55

4.2. Flow chart for insertions. 56

4.3. Example of insertion into the IF. Case 1. 57

4.4. Example of insertion. Case 2 on the left. Case 3 on the right. . . . 58

4.5. Number of intervals of different widths for Bayes, Kmeans, Labyrinth and Yada, both RS and WS (log scale). 60

4.6. Write set interval creation for maximum length transactions of Bayes, Kmeans, Labyrinth and Yada. 62

4.7. Execution time of Bayes, Kmeans, Labyrinth and Yada normalized to the perfect filter. 62

5.1. Effect of nullification of rows in the null space of an LS scheme's hash matrices ($N(H) \cdot H$ is the null matrix). 69

5.2. Pattern of aliases of zero for the LS matrices in Figure 5.1 (left), and its density distribution (right). 70

5.3. Probability of false positives of generic and LS signatures varying the parameter $f = \sum_{t=1}^{3} f_t$ (the higher the f, the more the locality). 71

5.4. Average Bloom filter occupancy saving of locality-sensitive signature with respect to generic signature, both for read set (top) and write set (bottom), varying the filter size from 64b to 8Kb. 73

5.5. Execution time normalized to perfect signature comparing parallel generic signatures and (r, δ)-LS-Sig with $r \in \{1, 3, 5\}$ and δ_0. 76

5.6. Execution time normalized to perfect signature comparing parallel generic signatures and (r, δ)-LS-Sig with $r \in \{1, 3, 5\}$ and δ_1. 78

5.7. Execution time normalized to perfect signature comparing parallel generic signatures and (r, δ)-LS-Sig with $r \in \{3, 5\}$ for δ_P and $r = 5$ for δ_1. 81

5.8. Average of RS and WS percentage of false positives for generic and $(3,\delta_P)$-LS signatures. 82

5.9. Number of false positives of parallel generic signatures (FP Generic) compared to the number of false positives of $(5, \delta_0)$-LS-Sig (top row), $(5, \delta_1)$-LS-Sig (middle row) and $(5, \delta_P)$-LS-Sig (bottom row). False positives for LS-Sig are broken down into false positives due to aliasing (FP) and false positives due to additional false sharing (FS). 83

5.10. Execution time normalized to perfect signature (no false positives) comparing Generic and $(r \in \{3, 5\}, \delta_P)$-LS-Sig to their PBX versions. 87

5.11. LS-Sig PBX matrices. A 1 in the topmost row and the leftmost column means that the 6th bit of the address is used to compute the 11th bit of the index. A 1 in the bottommost row and the rightmost column means that the 26th bit of the address is used to compute the LSB of the index. 88

6.1. Regular signature implementation schemes. 92

6.2. Parallel signature implementation schemes. 94

6.3. Multiset shared and reconfigurable asymmetric signature implementation schemes. 95

6.4. Asymmetric/multiset filter under analysis. 100

6.5. Side view and contour plot of the surface formed by the expected value of the false positive probability according to Expression 6.11. Part 1 of 3. 101

6.6. Side view and contour plot of the surface formed by the expected value of the false positive probability according to Expression 6.11. Part 2 of 3. 102

6.7. Side view and contour plot of the surface formed by the expected value of the false positive probability according to Expression 6.11. Part 3 of 3. 103

6.8. Explored solutions. 104

6.9. Execution time normalized to perfect signatures comparing regular and parallel separate signatures (SEP) to regular and parallel multiset ones (MS). 106

6.10. Average false positive percentage for regular SEP and MS signatures broken down into RS, WS and cross false positives (RS-X, WS-X). 108

6.11. Execution time normalized to perfect signatures comparing parallel separate, multiset and asymmetric signatures varying parameter *a*. 109

6.12. Percentage of addresses exclusively read, written and both read and written inside transactions. 110

6.13. Execution time normalized to perfect signatures comparing parallel separate and multiset signatures varying parameter *s*. 111

6.14. Breakdown of WS signature checks into negatives and positive matches (true and false) for all the benchmarks using multiset shared perfect signatures, which implies that false positives are due to read-read dependencies. 113

6.15. Percentage of Check RS+WS and Check WS in the execution of each benchmark using perfect signatures. Average percentage is also shown. 113

6.16. Execution time normalized to perfect signatures comparing parallel separate, locality separate and locality multiset $s=3$ signatures (L1 and L2). 114

6.17. Average speedup, per benchmark, of parallel MS $s=3$, MS $s=4$, MS $s=3$ L1 and MS $s=3$ L2 signatures over parallel separate signatures. 116

6.18. Average speedup, per signature size, of parallel MS $s\in[0,4]$, MS $s=3$ L1 and MS $s=3$ L2 signatures over parallel separate signatures. 116

6.19. Execution time normalized to perfect signatures comparing parallel multiset $s=3$ and parallel $s=3$ signatures. 118

7.1. Pseudocode of EigenBench core function. 123

7.2. Pseudocode of the function that generates the locality random walk. 125

7.3. Contention results for 15 threads. 128

7.4. Transaction length results for 15 threads. 130

7.5. Concurrency results for 15 threads. 132

7.6. Working-set results for 15 threads. N_{MILD} varies from 8K to 8M elements of 4 bytes. $R_{MILD} = W_{MILD} = 50$. The rest of parameters are set to 0. 133

List of Tables

2.1. An HTM taxonomy. 22

3.1. Base CMP system parameters. 43

3.2. STAMP benchmark suite qualitative characterization. 50

4.1. Parameters and data set maximum and average sizes. 59

4.2. Percentage of single addresses. 60

5.1. Example of locality-sensitive signature: addresses and their corresponding H3 indexes for a Bloom filter with k=4, 2^m=1024. 66

5.2. Probabilities defined by Expression 5.4 for STAMP codes (the higher the f_1 the more the spatial locality in transactions). 72

5.3. Input parameters for the benchmarks 74

5.4. Workload transactional characteristics. (Data set sizes in the rightmost four columns are reported in cache blocks) 75

6.1. Workload transactional characteristics. (Data set sizes in the rightmost columns are reported in cache blocks) 105

6.2. Implemented synopsys optimized XOR hash area versus estimated unoptimized upper bound area using 2 fan-in XOR gates (in μm^2). 119

6.3. Area (μm^2) figures for separate, multiset and asymmetric signatures, as well as time of the critical path (ns) and dynamic energy per access to both RS and WS (pJ). 65nm technology. $k = 4$ hash functions. 120

7.1. Parameters of EigenBench for the experiments. 130

7.2. Average read set and write set lengths ($|RS|$ and $|WS|$) measured by the HTM system compared to that of EigenBench input parameters. Transaction length is $|RS| + |WS|$, and the RS to WS ratio is $\frac{|RS|}{|WS|}$. 131

1 Introduction

With the shift to chip multiprocessors, hardware manufactures have put a big burden on software development community at all levels. Developers of operating systems, programming languages, applications, algorithms, data bases, etc. have to begin thinking in parallel and they must be aware of many details of the underlying hardware if they want to get the most out of chip multiprocessors. Researchers and hardware manufacturers are working to relieve software programmers of such a burden. For that purpose, they have devised transactional memory, a programming abstraction to hide low level hardware details from programmers. In this thesis, we focus on optimizing hardware transactional memory systems, which implement the transaction abstraction at the core level.

In next sections, we introduce the shift to multiprocessors (Section 1.1), which poses several problems for programmers that now have to deal with parallel programming and its complexity (Section 1.2), and then we discuss the transactional memory abstraction (Section 1.3), proposed to help on programming chip multiprocessors. Section 1.4 introduces the thesis motivation and contributions, and the thesis structure is outlined in Section 1.5.

1.1. The Shift to Chip Multiprocessors

Almost fifty years ago, Gordon E. Moore predicted that the number of transistors on a chip would double every two years [68]. Since then, the so-called Moore's law has been enforced by processor manufactures. Performance, though, has been more difficult to extract from processors in the last decade. Increasing clock frequency was used to gain performance until ten years ago, when power dissipation and cooling turned out to be a big concern [74]. Also, a significant part of the transistors on a chip has been devoted to implement hardware structures to extract instruction level parallelism (ILP) in the so-called superscalar pipelined processors. However, such techniques have proved to be limited by the amount of intrinsic parallelism of sequential applications [107], and alternative solutions have been proposed to harness superscalar processors, like simultaneous multithreading (SMT) [56] that fills the gap of ILP with thread level parallelism (TLP). With SMT processors, hardware industry hinted the paradigm shift towards non-automatically extracted parallelism. Finally, processor manufactures have shifted from large SMT superscalar single processors to thin-core, single-chip multiprocessors (CMPs) [35, 61] in order to deliver high performance computers and to keep pace with Moore's law in terms of performance as well.

CMPs have become mainstream in commodity processors, and thus, hardware industry has passed the baton of exploiting these parallel machines on to the software community. However, the majority of programmers are used to sequential programming, and multiprocessor programming is currently mastered by a small group. Parallel programming is more complex than programming uniprocessor machines and requires an understanding of new hardware concepts, algorithms, and programming tools. Thus, although hardware industry did not care about programmers when shifted to CMPs, hardware community is working on easing the task of programming CMPs [40].

1.2. The Complexity of Parallel Programming

Task parallelism is a common programming model for CMPs, where a parallel application is divided into separate threads of computations that run on different cores of the CMP. Writing multithreaded parallel programs, then, involves decomposing the problem we want to solve, assigning the work to computational threads, orchestrating such threads, and mapping them to the machine. Most of these steps are new for the programmer of sequential applications, and orchestration might be the most complex step, as it involves synchronizing the threads.

Parallelism introduces non-determinism that must be controlled by explicit synchronization. Shared data used in critical sections must be accessed in mutual exclusion to avoid race conditions between threads. CMPs usually provide special atomic instructions like compare-and-swap (CAS) or test-and-set (TS), that atomically execute a load and a store operation over a memory location and no other instruction can interleave between them. However, when the programmer wants to modify more than one shared memory locations at a time (also know as a critical section), CAS and TS instructions are not enough, and a mechanism to protect critical sections must be defined.

Lock-based techniques have been traditionally used to provide mutual exclusion and protect critical sections. A lock is a shared variable that is modified by using a CAS or a TS instruction so that just one thread can acquire it. Once a thread has acquired the lock, it can execute the critical section safely as other threads wait for the lock to be released. Hence, locks serialize the execution of concurrent threads in critical sections affecting the performance. Besides, locks introduce two sources of performance loss: *lock overhead* and *lock contention*. Lock overhead is the time needed to initialize and finalize the lock, the time needed to acquire and release the lock, and the memory space needed to allocate the lock. Lock contention is the time consumed by threads that are struggling to acquire the same lock. In general, there is a trade-off between lock overhead and lock contention, depending on the lock granularity. Coarse grain locking implies that a lock protects a large amount of data, so a few locks are needed throughout the application, and lock overhead is low. Lock contention will be high, though, and also serialization. On the other hand, finer lock granularity increases lock overhead, but lowers contention and serialization. Nevertheless, the risk of deadlock increases by narrowing critical sections, which makes it difficult to program and debug.

Locks may pose many problems that programmers should be aware of when programming parallel applications. *Deadlock* can occur if different threads acquire the same set of locks in different orders. It can be difficult to detect, particularly if the locks are not known in advance, which is the case of codes with indirections. Also, *convoying* can occur if a thread holding a lock is descheduled by the operating system, and other threads attempting to acquire such a lock are unable to progress. Finally, lock-based techniques lack effective mechanisms of abstraction and composition, as the programmer must know implementation details of library functions using locks to avoid the aforementioned problems.

1.3. The Transactional Memory Abstraction

Transactional Memory (TM) [40, 45, 54] emerges as an alternative to the conventional multithreaded programming to ease the writing of concurrent programs. TM introduces the concept of transaction, inherited from the database field, as a convenient abstraction for coordinating concurrent accesses to shared data, allowing semantics to be separated from implementation. A *transaction* is a block of computations that appears to be executed atomically and in isolation.

TM systems execute transactions in parallel, committing non-conflicting ones. A conflict occurs when a memory location is concurrently accessed by several transactions and at least one access is a write. In such a case, the conflict must be resolved by serializing conflicting transactions, so that atomicity is preserved. Unlike locks, in the absence of conflicts, transactions can run in parallel. Thus, transactions replace a pessimistic lock-based model by an optimistic one. Transactions also solve the abstraction and composition problems as semantics is separated from implementation and risk of deadlock or convoying is reduced due to the absence of locks. Besides, lock overhead and contention are no longer a problem, although new issues like livelock or other pathologies [9] can arise, but can be solved by the TM system to provide forward guarantees.

TM systems can be implemented in software (STM) [29, 39, 41, 44, 95] and hardware (HTM) [3, 37, 45, 69, 84]. There are hybrid and hardware accelerated software implementations as well [26, 53, 90]. STM systems implement transactions either by using locks or by utilizing non-blocking data structures [42]. However, STM implementations do not scale well and cannot compete with lock-based applications so far. However, HTM implementations provide most of the required TM mechanisms implemented in hardware at the core level, and they benefit from TM advantages without the overhead of STM. Hardware manufactures are introducing a form of HTM in their new CMP designs [17, 22, 87]. Hence, we focus on HTM systems in this thesis.

1.4. Thesis Motivation and Contributions

HTM systems must keep track of conflicts between transactions to ensure that atomicity and isolation are not violated. Early systems exploited the fact that conflicting requests to transactional memory locations could be observed through the cache coherence mechanism [14, 28, 37, 45, 69]. These implementations can support only transactions bounded in time and size, since a transaction is aborted when it overflows the hardware resources. More recently TM proposals include support for unbounded transactions, whose size and length are not constrained, by adding certain mechanisms to handle the overflow of hardware resources [3, 7, 8, 23, 48, 84, 103, 108, 110].

Many bounded and unbounded HTM systems use *signatures* to hold the addresses read and written by each transaction in order to detect conflicts [14, 59, 65, 67, 97, 110]. Signatures are usually implemented as Bloom filters [6], which are fixed hardware structures that summarize an unbounded amount of addresses at the cost of false positives. Such false positives can harm the performance seriously by provoking non-existing conflicts between transactions, particularly if transactions read and write large amounts of data. Although small transactions are assumed to be the common case [7, 24], several works give more insight into the “uncommon case” of large transactions to keep such an assumption from becoming a self-fulfilling prophecy [8, 111]. This thesis is motivated by the fact that HTM systems cannot easily cope with large transactions, and we contribute with proposals to optimize signatures and enhance the performance of applications that exhibit large transactions.

The main contributions of this thesis are the following:

- An *interval filter* that is proposed as an alternative to Bloom filters to track addresses read and written by transactions in form of contiguous memory location chunks, so-called intervals. Interval filters may show a lower false positive rate for those inserted elements that exhibit spatial locality according to a metric space. However, the rest of elements may be inefficiently managed. Hence, the interval filter is not a general solution.
- A *locality-sensitive signature* based on Bloom filters that exploits memory reference spatial locality. Previous signature designs consider all memory addresses as uniformly distributed across the address space. However, in real programs the address stream is not random as it exhibits some amount of spatial and temporal locality. Our proposal defines new maps for the hash functions of Bloom filters in order to reduce the number of bits inserted into the filter for those addresses with spatial locality. That is, nearby memory locations share some bits of the Bloom filter. As a result, false conflicts are significantly reduced for transactions that exhibit spatial locality, but the false conflict rate remain unalterable for transactions that do not exhibit locality at all. This is specially favorable for large transactions, that usually present a significant amount of spatial locality. In addition, as our proposal is based on new locality-aware hash maps, its implementation does not require extra hardware.
- A *multiset signature* that uses a single Bloom filter to track both read and write sets of transactions to deal with asymmetry in transactional data sets. Read and write signatures are usually implemented as two separate, same-sized Bloom filters. In contrast, transactions frequently exhibit read and write sets of uneven cardinality. In addition, both sets are not disjoint, as data can be read and also written. This mismatch between data sets and hardware storage introduces inefficiencies in the use of signatures that have some impact on performance, as, for example, read signatures may populate earlier than write ones, increasing the expected false positive rate. Multiset

signatures equalizes the false positive rate of read and write signatures and yield better results in most cases. Different alternatives were also studied to take advantage of some important properties of data access patterns, like either the significant amount of transactional memory locations that are both read and written, or the locality property.

- A *reconfigurable asymmetric signature* that can be dynamically configured at runtime to devote different amount of hardware either to the read set or to the write set of transactions. This signature design is an alternative to the common scheme, and to the proposed multiset signatures, to deal with asymmetry in transactional data sets in an effective way. We do not study a reconfiguration on a per-transaction basis. Instead we provide a heuristic to statically set the configuration of the signature at the beginning of the application, based on the average read set to write set ratio of its transactions.

- A *scalability study* is carried out to show the response of our proposals, compared to the common schemes, in the presence of contention, large transactions and different number of cores.

- A *thorough analysis and simulation* of all signatures proposed in this thesis is performed, including statistical studies of false positives and a full evaluation of signature performance using state-of-the-art simulation tools and benchmarks used widespread in TM literature.

The aforementioned contributions have been published in international peer reviewed conferences [79, 81, 82, 83], workshops [80] and journals [77, 78] ranked by the ISI Journal Citation Reports (JCR).

1.5. Thesis Structure

The remainder of this thesis is structured in the following way:

- *Chapter 2* introduces the basics of transactional memory focusing on its semantics. Next, an overview of the main TM implementations proposed in the literature is presented, along with a description of the HTM extensions that hardware manufactures are planning to deploy on their forthcoming CMPs. Finally, we discuss related work on signature conflict detection.

- *Chapter 3* outlines the methodology we have followed to evaluate our proposals. We describe the simulation tools and the benchmark suite chosen for experimentation.
- *Chapter 4* deals with interval filters, an alternative to Bloom filters, that is a locality-aware approach that we propose to tackle large transactions that exhibit some amount of spatial locality.
- *Chapter 5* presents locality-sensitive signatures as an enhancement of Bloom filters in the line of harnessing the spatial locality property of applications, without adding hardware complexity.
- *Chapter 6* describes the multiset and reconfigurable asymmetric signature proposals to deal with asymmetry that most transactional applications usually exhibit in their data sets.
- *Chapter 7* studies the scalability of the proposed signature schemes in terms of contention, transaction size and speedup.

The last section concludes the thesis and suggests certain possible lines for future work.

2 Background and Related Work

This chapter presents a background on transactional memory (TM) starting from an overview of its semantics and requirements in Section 2.1. Section 2.2 focuses on various implementations of TM systems, both software and hardware, and summary current research on the TM topic. New TM extensions for CMPs of main hardware manufacturers are reviewed as well in this section. Finally, this chapter includes an overview of related work on signature conflict detection in TM (see Section 2.3).

2.1. Transactional Memory Basics

In this section, we outline the semantics and the basics of TM from the perspective of the programmer (see Sections 2.1.1, 2.1.2 and 2.1.3). Also, Sections 2.1.4 and 2.1.5 discuss different mechanisms required for implementing a transactional system that complies with the semantics presented in Sections 2.1.1 to 2.1.3. This section avoids discussing implementation specifics which can be found in Section 2.2.

2.1.1. Programmers' View

From the programmers' perspective, TM is a programming abstraction rather than a whole system to support optimistic concurrency in parallel applications. Programmers are abstracted from all the implementation mechanisms of TM and they are provided with simple language extension constructs and clear semantics.

Figure 2.1 shows various language constructs for delimiting transactions. In the code on the left, the *atomic* statement is used to point out that the enclosed instructions must be executed transactionally. We can also define an entire function as atomic. From the programmers' view, an atomic block or function executes all or nothing. That is, either all instructions are executed successfully, in which case the transaction *commits* and its results become visible to other transactional and non-transactional code, or the transaction *aborts* leaving the program's state unchanged.

The semantics of transactions stem from database systems where the ACID (Atomicity, Consistency, Isolation and Durability) set of properties applies. In the case of TM, durability is not important since transactions work with transient main memory data. Regarding consistency, a transaction should change the state of the system to a different consistent state, however, the meaning of consistency is application dependent and it must be specified by the programmer, so the system must provide the means for the programmer to ensure consistency (see Section 2.1.2). Finally, the TM system must ensure that transactions comply with the rest of the properties:

- *Atomicity* implies that a whole transaction is executed as if it is a single, indivisible instruction.
- *Isolation* means that changes made inside transactions do not affect the program's state until they have successfully committed.

With TM, programmers do not need to define a shared variable (lock) to protect the access to other shared variables. Thus, TM allows *composition*, a desirable feature for software productivity and programmability. For example, function `bar()` in Figure 2.1 could be part of a library using shared resources. In case of using transactions, the code of `bar()` is executed as part of the atomic block. If locks are used, the programmer must be aware of how `bar()` implements the access to shared resources to avoid possible deadlocks due to nested locks. However, TM can deal with nested transactions, which might result of composition or can be explicitly declared by programmers. Next section discusses the semantics of nesting alternatives in TM.

2.1.2. Nested Transactions

Transaction nesting is usually supported by TM systems to facilitate software composition, as well as to increase parallelism and expressiveness. A nested

```
atomic {
  if (x != 0)
    y = y + (x = bar());
}

atomic void foo {
  if (x != 0)
    y = y + (x = bar());
}
```

```
...
atomic {
  if (x != 0)
    y = y + x;
  atomic closed {
    x = bar();
  }
}
...
```

```
...
atomic {
  if (x != 0)
    y = y + x;
  atomic open {
    x = bar();
  }
}
...
```

Figure 2.1: TM language extension constructs for programmers.

transaction, also known as inner transaction, is a transaction entirely enclosed by another transaction, called the outer transaction. The easiest way for TM systems to deal with nested transactions is to flatten them so that they are part of the outer transaction. This, although maintain correctness and composability, can lead to performance degradation since an abort in the inner transaction implies aborting the outer transaction as well. TM systems may support two nested transaction alternatives to enhance parallelism and expressiveness, the so-called closed and open nested transactions [71]. Their semantics are described below:

- *Closed nested transactions*: A closed transaction can abort without aborting the outer transaction. In case of commit, the outer transaction can see the changes made by the inner transaction. However, other transactions running in the system are unable to see the changes until the outer transaction has successfully committed. The code in the middle of Figure 2.1 shows a language construct for closed nested transactions. In the example, if the execution of `bar()` aborts, the computations of the outer transaction are not wasted.

- *Open nested transactions*: Open transactions add more expressiveness to TM systems and can improve program performance. An open transaction releases isolation of data on commit. That is, its changes become visible not only to the outer transaction but also to all other transactions running in the system. Even if the outer transaction aborts later on, the changes made by open nested transactions persists. On abort, open transactions behave like closed ones. The rightmost example in Figure 2.1 shows a language construct for open transactions. They are useful for enhancing parallelism in certain situations like allowing garbage collectors, or for library codes. However, open transactions might alter the consistency of the system, for example, if a call to `malloc()` was executed by the inner transaction and the outer transaction eventually aborts. So, the TM system must provide the means for the programmer to ensure consistency. TM systems usually allow the definition of compensating actions (it would be a call to `free()` in the given example) to be executed in case of the outer transaction aborts.

Another construct to enhance transaction parallelism can be supported by some TM systems: early-release [44]. Early-release allows a transaction to remove addresses from its transactional read-set before commit. Thereby, other transactions can write to those addresses without generating a conflict with the releasing transaction. Early-release is similar to open transactions, but it only applies to reads. Also, the programmer must guarantee that early-release does not violate the overall application atomicity and consistency, which rises the complexity of using early-release to be similar to that of using fine-grain locks [98].

2.1.3. Weak and Strong Isolation

When using transactions, programmers must be aware of how the TM system tackles the interaction between transactional and non-transactional code. Two different types of isolation can be defined in TM systems [62]: weak and strong isolation.

Weak isolation guarantees isolation between transactions. However, non-transactional code can access data written inside transactions without protection from the TM system, thus breaking atomicity and isolation of transactions. This way, non-transactional accesses may introduce data races on transactional data that could lead to undefined behaviour of the system. Then, to prevent undesirable inconsistencies, programmers should explicitly protect every access to shared data with a transaction. Another option is to separate transactional from non-transactional code by means of barriers, but it can degrade performance and scalability.

On the other hand, *strong isolation* guarantees isolation between transactions and between transactional and non-transactional code. With strong isolation, instructions outside transactions automatically become single-instruction transactions. This fact implies that atomicity and isolation are enforced in every situation. However, consistency could not be always guaranteed since a programmer might incorrectly delimit the bounds of transactions. Unfortunately, controlling whether or not programmers properly place the boundaries of transactions is out of control of the transactional system.

2.1.4. Data Versioning

To enforce TM semantics introduced in preceding sections, TM systems must implement certain mechanisms of *data versioning*. Data versioning, also known as version management, implements the policies that tells the TM system what to do with data modified inside transactions. Two policies can be considered:

- *Eager data versioning*: With eager version management transactions modify data in place with the risk that other transactions can see the new versions of the data, thus violating isolation. In this case, the data versioning policy must be supported by the conflict detection system (see Section 2.1.5) to ensure TM semantics by controlling the access to shared data. On abort, eager versioning systems must restore the old values of data modified by the aborting transaction. Therefore, a private segment of memory, also known as *undo log*, is needed to keep track of those old values. Commits, though, are faster in eager data versioning systems since new values are already in place.

- *Lazy data versioning*: Lazy version management consists in leaving old values in place while maintaining new values in private memory. Then, the TM system must ensure that a transaction always accesses the last updated value of its data. With lazy versioning, isolation is guaranteed and conflict detection is in charge of enforcing atomicity. This policy also needs a private write buffer, or *redo log*, to store new values. Aborts are faster now, since the system only needs to discard the new values in the write buffer. Commits are slower, though, as lazy data versioning systems must update memory data with the new values stored in the write private buffer of transactions.

Some subtleties can be exposed when dealing with data versioning, granularity of data versioning and weak isolation described in the preceding section. Let's give an example: suppose a TM system implementing the eager version management policy at cache line granularity, that is, whenever a transaction modifies one word of a cache line the TM system stores the whole line as it was before the modification, just in case of a potential abort, to restore the old value. First, a transaction writes a word of a cache line, and next, a piece of non-transactional code modifies a different word of the same cache line, which do not implies a real conflict. Then, the transaction happens to abort, thus undoing the modifications made. Since data versioning granularity is cache-line-wise, the old value of the entire cache line is restored, and non-transactional work is undone causing unpredictable behaviour. These subtleties can be difficult to debug when programming a TM system and suppose a challenge for weak isolation systems.

The problem outlined above can be avoided in systems that enforce strong isolation, as non-transactional instructions are managed as if enclosed by single-instruction transactions and isolation is guaranteed. However, there exists the possibility of these kind of data races in strong isolation TM systems if granularity of data versioning is coarser than granularity of conflict detection. This way, in the example above, the conflict would not be detected and the work could be undone in case of abort.

2.1.5. Conflict Detection and Resolution

A TM system requires a mechanism for conflict detection and resolution to ensure atomicity, and some times isolation as we saw in the preceding section. In a CMP with multiple threads running concurrently, the memory system sees instructions from different threads interleaving each other arbitrarily. A bunch of instructions can execute atomically by serializing the execution and avoiding such interleaving, but it degrades performance. However, TM is able to allow thread interleaving while it controls atomicity by means of conflict detection and resolution, thus maximizing the opportunities for thread level parallelism (TLP).

A *conflict* is detected when more than one transaction accesses the same memory location and at least one of them modifies it. A conflict might be detected as well in case that conflict detection granularity is coarser than one memory location. Then, false conflicts due to false sharing can arise.

Conflict detection can be implemented following two different approaches, the same way as data versioning in Section 2.1.4: eager or lazy conflict detection.

- *Eager conflict detection*: If conflicts are detected eagerly, the TM system must keep track of every transactional access "on-the-fly", so that the conflict is detected just before it occurs. This way of detecting conflicts is also known as pessimistic (pessimistic within the optimistic framework of TM) because it supposes that conflicts frequently arise and must be detected on arising to keep transactions from working with stale data.

- *Lazy conflict detection*: With lazy conflict detection, the TM system allows transactions to access shared data concurrently, while conflict detection is deferred to commit time. Notice that transactions should work in isolation with the last valid version of the data. This kind of conflict detection is said

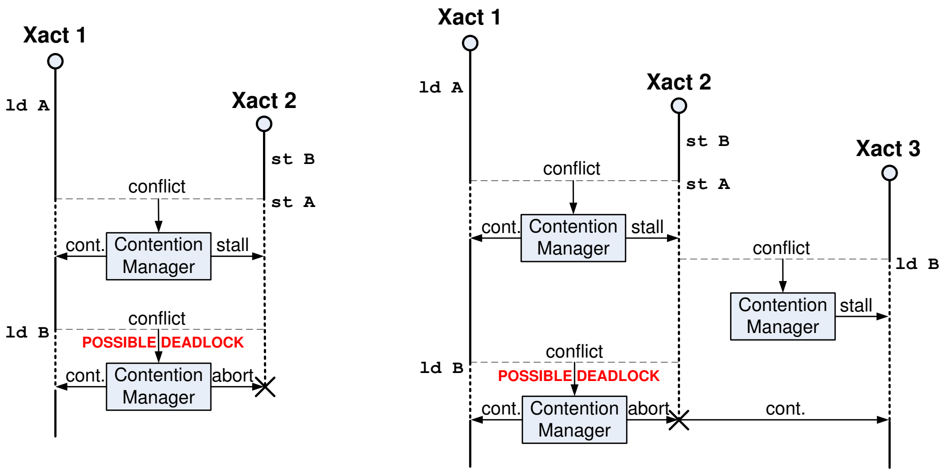


Figure 2.2: Examples of possible deadlock situations in TM systems.

to be optimistic, since it encourages parallelism if conflicts are infrequent. Conversely, if conflicts are often encountered, lazy conflict detection could readily serialize execution, since the TM system would abort those transactions that worked with stale data. Eager conflict detection mechanisms would ameliorate the amount of wasted execution in such a case.

A TM system needs to know which memory locations have been accessed by each transaction to detect conflicts and then resolve them. To keep track of such accesses, each transaction usually owns two sets: the *read set* (RS), which holds the address of every location read by the transaction; and the *write set* (WS), holding the addresses written by the transaction. The implementation of these sets depends on whether the TM system is software or hardware implemented. For hardware systems, RS and WS storage supposes a problem since an unbounded number of addresses (transaction size is not known in advance unless the system restricts the size of them) must be recorded in a fixed-size structure.

Once a conflict is detected, the TM system has to resolve it. This is called *conflict resolution* and is typically implemented by a *contention manager*, whose aim is ensuring forward progress and avoiding deadlock situations. There are two ways of resolving a conflict: (i) to abort all transactions involved except one of them, in which case a great amount of computation might be discarded; and (ii) to let one transaction continue whereas the others stall, waiting for the running transaction to commit and release isolation of the conflicting data.

Stalling transactions can reduce the number of wasted cycles but it involves the risk of deadlock.

Figure 2.2 shows two different situations of possible deadlock when resolving conflicts using stalls. In the example on the left, transaction 1 begins and then it reads location A. After a while, other thread begins a transaction, Xact 2, that modifies location B and then, tries to write location A. At this point, a conflict is detected and the contention manager decides to stall the "youngest" transaction, Xact 2. Xact 1 is allowed to continue. Then, Xact 1 happens to read location B that was previously written by Xact 2. A conflict is detected, and stalling Xact 1 would cause a deadlock situation. Hence, the contention manager must abort one transaction and it chooses Xact 2. The example on the right shows an scenario involving three transactions. Xact 1 and Xact 2 exhibit the same dependencies as in the former example, however, another transaction enters the scene, Xact 3, which is stalled because of a dependency with Xact 2. Then, when Xact 1 conflicts with Xact 2, a threefold conflict arise and the contention manager has to decide whether to abort Xact 2 or Xact 3 to prevent the possible deadlock.

Example scenarios in Figure 2.2 seem to be trivial to resolve, but they can complicate more in the presence of multiple transactions. Contention managers might need much information from the system, like timestamps to discern how long transactions have been running, a list of stalling transactions and a list of dependencies between transactions. Also, different conflict resolution policies can be implemented [94] and changed across workloads, across concurrent transactions in a single workload, or across different phases of a single transaction to fit workload characteristics [36]. Thus, a contention manager might suppose too much state to implement in hardware, so several hardware systems trap to software contention managers to handle conflicts (see Section 2.2.2).

2.2. Overview of Transactional Memory Implementations

The preceding section deals with transactional memory from the point of view of its semantics and requirements. In this section, we review the main TM implementations that can be found in the literature, both software and hardware, showing which policies were chosen in each implementation from those discussed in the last section. Software transactional memory (STM) will be briefly discussed in Section 2.2.1, whereas we will give more insight into hardware transactional memory (HTM) in Section 2.2.2, since we focus on HTM systems in this thesis.

2.2.1. Software Transactional Memory

The term software transactional memory (STM) was introduced by Shavit and Touitou in [95], where they proposed a STM system that provided a transaction abstraction approach slightly different from that we have seen in Section 2.1. They required the programmer to provide in advance the set of variables that a transaction might access. Thus, the system would acquire ownership of them so that the transaction could execute until commit without the possibility of roll-back.

Many proposals have been developed since Shavit and Touitou first proposed STM. The majority of them do not require a programmer to specify in advance the locations that a transaction is going to access. However, these systems seem to introduce overhead to such an extent that STM has been said to be a toy for researchers [13]. Nevertheless, research on STM is still ongoing due to its advantages over HTM systems, namely, its flexibility on implementing a full range of sophisticated algorithms, its facility to be modified and integrated with existing systems, and the absence of fixed-size hardware structures that might impose limitations.

STM systems are often implemented as runtime systems that profile and instrument the source code to provide the functionality of a TM system. Such runtime systems can be implemented using locks to provide atomicity. We will see later other nonblocking STM systems that do not use locks in their implementation. *Lock-based* STM systems like Bartok-STM [41], Multicore RunTime STM (McRT-STM) [89], Transactional Locking 2 (TL2) [29] and TinySTM [32] use two-phase locking from databases [30] to avoid deadlocks. Locks, which can be associated to data on a per-object or a per-word basis, are acquired in one phase, as transactions write data or when commit begins, an no locks are released until the shrink phase, at commit end, in which locks are released and no locks can be acquired.

Bartok-STM and McRT-STM implement eager version management to avoid log lookups on all read operations, since searching transaction logs for earlier stores by the same transaction does not scale in case of large transactions. Conflict detection is also carried out eagerly, except for reads, which are validated lazily on commit. Thus, transactional read overhead due to conflict detection is kept low by just logging the addresses and some metadata for those data being read. Therefore, as the read set of transactions usually outnumbers the write set (see Chapter 6), additional contention in the memory system is avoided and conflict-free transactions stay as simple and fast as possible.

While conflict detection on concurrent writes are arbitrated by locks, detecting a read-write conflict requires the use of *versions* when reads are detected lazily. *Versioned locks* combine a conventional lock for writes and an integer variable for reads, so-called version. The version of the lock is incremented whenever it is released, which means that the data protected by the lock was successfully updated. Then, transactional reads store into the read set the version number of the lock protecting the data, and eventually, on commit, all versions in the read set are checked against the current versions of the locks. If one version number do not match, a read-write conflict is detected and has to be resolved. McRT-STM and Bartok-STM use a version number incremented on a per-object basis whereas TL-2 uses a *global clock*, which is incremented each time a transaction commits.

TL-2 uses lazy version management, so updates are held in private memory. Then, versioned locks of data updated in transactions are acquired at commit time. If one lock cannot be acquired the transaction fails, as a write-write conflict is detected. Next, the read set is traversed to check the version number of each location read against the global version clock acquired when the transaction began. If one version number is greater than the global clock, a read-write conflict is detected and the transaction fails.

TinySTM is similar to TL-2 but uses eager version management and implements a form of hierarchical locking in which version numbers are hold at slot granularity, with such slots covering regions of the entire address space. These slots are implemented as a shared array and hold the number of commits to locations in the region represented by the slot.

On the other hand, *nonblocking* STM systems, like Herlihy's Dynamic STM (DSTM) [44], Object-based STM (OSTM) [34] or Word-based STM (WSTM) [39], provide progress guarantees, atomicity, and isolation in the absence of locks by using non-blocking algorithms performing single-word atomic operations. DSTM is object-based. A transaction is defined as an object with function members for beginning and committing it. DSTM also requires that objects accessed by transactions are wrapped by a TMObject class. This class has a pointer to a *locator* class whose fields are three pointers pointing to the last transaction that opened the object in write mode, the old value for the object, and the new value for the object. Then, when a transaction writes an object, it changes the locator pointer of the object to point to a new locator object. In turn, the new locator points to the transaction that writes the object. Indirections make atomicity possible by using an atomic CAS to change pointers of locators. DSTM uses eager conflict detection both on reads and writes.

OSTM is similar to DSTM. However, OSTM uses lazy conflict detection, and a TM object refers to a transaction descriptor instead of using a locator. As a nonblocking system without indirections we have WSTM, which is a word-based STM system. Thus, data is normally stored in the heap, instead of being wrapped by a transactional object, and transactional information is associated to each datum in the heap by means of a hash function. WSTM uses lazy conflict detection.

STM systems usually *decompose* transactional read and write operations into several stages segregating conflict detection, data versioning and data accessing. This decomposition allows compiler optimizations like instruction reordering or redundancy elimination.

Finally, the majority of STM proposals provide a programming abstraction that is slightly different from the atomic block abstraction described in Section 2.1.1. The programmer must ensure that every read and write operation, or every object used in transactions, is tagged as transactional, and it can be tedious and unattractive for the programmer. Several works [41, 43] point out that the atomic block abstraction is easily achievable by a compiler, which can automatically generate the required code.

2.2.2. Hardware Transactional Memory

Hardware transactional memory (HTM) systems have several advantages over STM systems. HTM has lower overheads, is less dependent on compiler optimizations, and more transparent to the programmer than STM, all memory accesses in a transaction are implicitly transactional and third-party libraries do not need to be transaction aware. In addition, HTM systems can provide strong isolation warranties without requiring changes to non-transactional code, and can have better power and energy profiles.

Next, we present the first HTM proposal that was the base for subsequent HTM systems. Then, we discuss the main HTM proposals we can find in the literature, and, to conclude, we deal with commercial CMP's that are including HTM extensions as a feature for enhancing concurrency.

First HTM Approach

HTM, and the TM abstraction, was first proposed by Herlihy and Moss [45] in 1993 as short transactions intended to replace short critical sections. They introduced changes to three levels of a CMP architecture:

- *Processor level*: Each processor maintains two flags, one to indicate whether the processor is executing transactional or non-transactional code (a *transaction active* flag), and another to indicate the transaction *status* (ongoing or aborted) whenever the active flag is set. The transaction active flag is implicitly set when a transaction executes its first transactional operation. Also, the processor maintains a private *transactional cache*, besides the private primary cache. The transactional cache is a small full-associative cache that holds transaction updates, both old and new versions. The primary cache and the transactional cache are exclusive so an entry may reside in one or the other, but not both.

- *ISA level*: Six new instructions are added to the instruction set architecture to support transactions: a transactional load (`LT`), a transactional load in exclusive mode (`LTX`) hinting that the location to be accessed is likely to be updated, a transactional store (`ST`) to write a location into the transactional cache, and `COMMIT`, `ABORT` and `VALIDATE` instructions. On commit, the status flag is checked and returned to the user. If it is set to true, new versions in the transactional cache are tagged as non-transactional, and old versions are discarded. The transaction active flag is set to false as well. If the status flag is set to false, then the transaction must abort because a conflict was detected during its execution. Thus, old versions are tagged as non-transactional, new versions discarded and the transaction active flag is set to false. Moving data from the transactional cache to the primary cache is not needed as the transactional cache is a part of the memory hierarchy and snoops the network.

- *Cache coherence level*: The cache coherence protocol is modified by adding three more messages. One for requesting a location because of a transactional load, other one for requesting a location because of a transactional exclusive load, and a busy message to signal conflicts. When a transaction loads a location, its transactional cache is searched in case the location was previously written by the same transaction. In case of a miss, the value is requested to other processors. Such processors check their transactional caches (this is performed in one cycle as caches are associative). If at least one processor hits its transactional cache, a busy message is sent to the requesting processor. Then, the requesting processor sets its status flag to false (aborted) and subsequent transactional loads and stores do not cause network traffic and may return arbitrary values. Therefore, conflict resolution can be said to be eager, although the conflict does not resolve until the program executes a commit/abort/validate instruction that checks the status flag.

```
typedef struct {
  Word deqs;
  Word enqs;
  Word items[QUEUE_SIZE];
} queue;

unsigned queue_deq(queue *q) {
  unsigned head, tail, result, wait, backoff = BACKOFF_MIN;
  while (1) {
    result = QUEUE_EMPTY;
    tail = LTX(&q->enqs);
    head = LTX(&q->deqs);
    /* queue not empty? */
    if(head != tail) {
      result = LT(&q->items[head % QUEUE_SIZE]);
      /* advance counter */
      ST(&q->deqs, head + 1);
    }
    if (COMMIT()) break;
    /* abort => backoff */
    wait = random() % (01 << backoff);
    while (wait--);
    if (backoff < BACKOFF_MAX) backoff++;
  }
  return result;
}
```

Figure 2.3: Example of programming Herlihy and Moss' HTM system. Fragment of a producer/consumer benchmark [45].

Figure 2.3 shows a fragment of a producer/consumer benchmark that uses Herlihy's HTM system. The transaction begins whenever the program executes instruction `tail = LTX(&q->enqs)`. If a conflict is detected when that instruction is executed, the status flag of the processor is set to false (aborted), and subsequent instructions will not produce valid data. Then, when the transaction executes the `COMMIT()` instruction, the program realizes that there was a conflict and resolves the conflict by aborting the transaction. In this case, abort actions consist in an adaptive back-off to reduce contention, and a retry of the transaction. Subsequent HTM systems usually provide the simple *atomic* programming abstraction of Section 2.1.1, instead of the explicit transactions proposed by Herlihy and Moss.

		Data Versioning	
		Lazy	**Eager**
Conflict Detection	**Lazy**	Stanford TCC Urbana-Champaign Bulk	—
	Eager	MIT LTM Intel/Brown VTM	MIT UTM Wisconsin LogTM Wisconsin TokenTM

Table 2.1: An HTM taxonomy.

Herlihy and Moss proved that transactions outperformed conventional locks because no memory accesses were required to acquire and release locks. Also, they pointed out how optimistic concurrency of transactions improved the performance over locks serialization. However, Herlihy's HTM system posed certain drawbacks. If transactions run for too long, they could be aborted by interrupts or conflicts. Also, the larger the data set of transactions, the larger the transactional cache needed, which can be prohibitive as far as it is a full-associative cache, and the more likelihood of conflict as well.

Herlihy's HTM lays the foundations for succeeding HTM systems. Most of them introduce changes at the three aforementioned levels of a CMP and try to solve the drawbacks of the system regarding transaction length, interrupts and hardware limitations. Next, we will see main HTM systems after Herlihy's and how some of them try to virtualize resources to hide HTM limitations to programmers.

Main HTM Proposals

Table 2.1 shows an HTM taxonomy of main HTM systems depending on conflict detection and data versioning policies. Let us see the continuation of the work of Herlihy, called Virtual TM (VTM) [84]. VTM comprises two parts, one purely hardware part which resides in processor local buffers and provides fast transactional execution for common case transactions that do not exceed hardware resources and are not interrupted. And a programmer-transparent overflowed part that resides in data structures in the application's virtual memory combined with hardware machinery, that allows transactions to survive overflow, page faults, context switches, or thread migration.

VTM assigns each transaction a status word (XSW), which is used to commit or abort the transaction by modifying it atomically with a CAS instruction.

VTM also defines a transaction address data table (XADT), which is the shared log for holding overflowed transactional data. Both structures reside in the application's virtual address space. However, they are invisible to the user. The VTM system, implemented in either hardware or microcode, manages these structures by means of new registers added to each thread context that point to them and are initialized by the application. When a transaction issues a memory operation that is a cache miss, it must be checked against overflowed addresses by traversing the XADT. Traversing the XADT might be too slow, so VTM provides two mechanisms for not interfering with transactions that do not overflow. First, an XADT overflow counter records the number of overflowed entries. If it is set to zero, no traffic is needed as it is locally cached at each processor. Second, an XADT filter (XF), implemented as a software counting Bloom filter (see Section 2.3.5), provides fast detection of conflicts. A miss in the filter guarantees that the address does not conflict, and a hit triggers an XADT walk.

Large TM (LTM) [3] is an eager-lazy HTM system like VTM, but slightly differs from VTM in the way it manages cache evictions. In LTM, each cache block is augmented with a transactional bit and each cache set with an overflow bit. If a processor requests a transactional block whose overflow bit is set, then the block will be in a hash table in uncached main memory. Such a hash table structure is managed by a hardware state machine transparently to the user. The operating system allocates and resize the structure as required by the hardware state machine which holds a register pointing to the hash table structure. Unlike VTM, LTM only allows transactions that fit in main memory and do not run longer than a scheduler time-slice.

On the lazy-lazy cell in Table 2.1 we can find Transactional Coherence and Consistency (TCC) [37]. This proposal defines a new coherence and consistency model in which transactions are the basic unit of parallel work. Writes within a transaction do not require coherence requests as they are locally stored in a write buffer. Regarding consistency, all memory accesses from a processor that commits earlier happen before the memory accesses of processors that commits later, regardless of if such accesses actually interleaved each other. TCC relies on broadcast to send the write buffer of a transaction to all processor so that they can check if there were a conflict. Local caches have transactional read and write bits for conflict detection. Also, unordered and ordered commit is allowed due to the commit control system which commits transactions depending on their phase number. Transactions with the same phase number can commit in any order, while transactions with a lower phase number must commit earlier than transactions with a greater phase number. Bulk HTM [14] also relies on broadcasting to detect conflicts. However, it only sends the addresses of the locations written by the transaction, which are stored in a local and compact way in form of *signatures*. Signatures were first proposed in Bulk HTM as Bloom filters (see Section 2.3). Bulk HTM uses caches to hold new data versions transparently to the cache itself. TCC and Bulk allows larger transactions than Herlihy's first HTM system but they do not solve the overflow problem of write buffers and local caches. Broadcasting is a feature that limits scalability as well.

On the eager-eager side, we have Unbounded TM (UTM) [3]. UTM is the base for LTM, however, UTM allows transactions whose memory footprint can be nearly as large as virtual memory, like VTM. UTM holds, in virtual memory, a structure called XSTATE which represents the state of all transactions running in the system. Besides, each memory block is augmented with a transactional read/write bit and a pointer to the old value of the block that resides in an entry of the XSTATE structure. Such an entry of the XSTATE structure, in turn, has a pointer to the memory block. So, the XSTATE structure holds a linked list of memory blocks whose transactional read/write bits are set. Conflict detection is carried out eagerly, so every memory access operation must check the pointer and bits of the memory block to detect any conflict. The access to the XSTATE and memory block metadata is done by means of several hardware registers that hold pointers to their base and bounds. For non-overflowed transactions, UTM implements a conventional cache-based HTM for performance.

LogTM [69] uses per-thread software-resident logs for version management instead of hardware buffers or caches like TCC and Bulk. Like in UTM, LTM and VTM, version management is held in virtual memory. However, UTM and VTM add significant complexity to their global logs and LTM's log is not cacheable, so traversing such structures is too expensive in terms of performance. LogTM optimizes the access to the logs by caching their entries. Also, only old values are stored and LogTM uses hardware structures to mask such writes and to prevent multiple buffering of the same location. LogTM is leveraged to support virtualization by decoupling transactional state from caches in its signature edition version, called LogTM-SE [110]. Support for nested transactions [70] is also included. Wisconsin's group proposed a different approach of HTM, called TokenTM [8], in which transactional metadata is not only used for overflows like in VTM and UTM. Instead, each memory block is augmented with a number of tokens. A transaction must acquire at least one token to read the block and must have all tokens to write it.

Many works propose enhancements to the baseline systems discussed above and give more insight into HTM. Bobba et al. [9] shows a set of possible pathologies that can harm the performance of HTM systems, and propose conflict resolution policies to solve them. Waliullah and Stenstrom [106] propose starvation-free protocols for transactions in TCC-like systems. Ramadan et al. propose DATM [85] to commit conflicting transactions if the conflict can be solved by forwarding data between them. Pant and Byrd [76] use value prediction to enhance parallelism, achieving an effect similar to that of forwarding. Lupon et al. [58] propose fast recovery aborts for transactions in LogTM-like systems by maintaining non-speculative values in the higher levels of the memory hierarchy. Waliullah et al. [105] also propose enhancements in speculative data management. Titos et al. [102] propose a hybrid-policy HTM system which can switch from eager to lazy version management depending on data contention. Negi et al. [73] improve scalability of lazy data versioning systems whose burden resides in broadcasts. Finally, we can find several hybrid approaches that enhance STM by hardware accelerating certain TM mechanisms [5, 53, 55, 67, 90, 96, 97].

HTM in Commercial Off-The-Shelf Multiprocessors

Main hardware manufacturers are introducing HTM extensions into their new CMP systems. Sun Microsystems has been working on the Rock chip-multithreading processor (CMT) [17], the first to support HTM. Each Rock processor has 16 cores, each configurable to run one or two threads. Then, a single core can run two application threads, giving each thread hardware support for executing ahead (EA) under long-latency instructions, like a data cache miss or a TLB miss. When the core encounters one of these long-latency instructions, a checkpoint of the architectural state is taken, the destination register of the long-latency instruction is marked as not available (NA) in a dedicated NA register file, and the long-latency instruction goes into a deferred queue (DQ) for later reexecution. Every subsequent operation with an operand marked as NA is enqueued in the DQ. Instructions whose operands are available are executed, and their results are stored in the speculative register file. When the long-latency operation completes, a replay phase is performed in which the deferred instructions are read out of the DQ and reexecuted. A different execution mode devotes core resources to run one application thread with even more aggressive simultaneous speculative threading (SST), which uses the second thread to execute in parallel the replay phase of the EA. EA is an area-efficient way of creating a large virtual issue window. So, Rock supports HTM by using most of the mechanisms that enable EA. In addition, two new instructions have been added to the instruction set: `checkpoint fail-pc` to denote the beginning of a transaction, which accepts a pointer to compensating action code used in case of abort, and `commit` to denote the end of the transaction. Also, cache lines include a bit to mark lines as transactional. Invalidation or replacement of a cache line marked as transactional aborts the transaction. Stores within the transaction are placed in the store queue and sent to the L2 cache, which then tracks conflicts with loads and stores from other threads. If the L2 cache detects a conflict, it reports the conflict to the core, which aborts the transaction. When the commit instruction begins, the L2 cache locks all lines being written by the transaction. Locked lines cannot be read or written by any other threads, thus ensuring atomicity. Rock's TM supports efficient execution of moderately sized transactions that fit within the hardware resources.

AMD is working on its Advanced Synchronization Facility (ASF) [22], an AMD64 architecture extension for HTM support. ASF adds six new instructions to the ISA: `speculate` denotes the beginning of a transaction and takes a checkpoint of the thread context for recovery, `lock mov` is used to load and store transactional data, `watchr` and `watchw` start a conflict detection operation for the operand, `release` can be used as the early-release construct discussed in Section 2.1.2, `commit` ends the transaction and makes all speculative modifications instantly visible to all other CPUs, and `abort`, that also ends the transaction and discards the modifications. These ISA extensions provide a transactional abstraction similar to that of Herlihy and Moss' HTM, in which transactional accesses must be explicitly annotated. However, AMD's compiler [22] is able to provide the conventional *atomic* abstraction for C/C++, while it annotates instructions transparently to the user (stack known local variables may not be annotated as transactional to save hardware resources). ASF implements a straightforward requester-wins conflict resolution policy, which always aborts the transaction already containing the conflicting element in its working set, and provides strong isolation. AMD has explored several implementations of ASF: a cached-based implementation that keeps the transactional data in each CPU core's L1 cache and uses the regular cache-coherence protocol for conflict detection, and an implementation that introduces a new CPU data structure called the locked-line buffer (LLB). The LLB is implemented as a full-associative memory similar to Herlihy and Moss' *transactional cache*. The advantage of an LLB-based implementation is that the cache hierarchy does not have to be modified. A hybrid approach combining the cached-based implementation and the LLB is also explored.

Intel has released details of its Transactional Synchronization Extensions (TSX) [87] for the future multicore processor code-named *Haswell*. TSX provides two interfaces to denote transactional code. The first one is known as Hardware Lock Elision (HLE), and involves two prefixes for instructions: `XACQUIRE` and `XRELEASE`. HLE is compatible with the conventional lock-based programming model. So, software written using the HLE prefixes can run on both legacy hardware without TSX and new hardware with TSX, since the prefixes correspond to the `REPNE`/`REPE` IA-32 prefixes which are ignored on the instructions where `XACQUIRE` and `XRELEASE` are valid. Thus, the programmer uses the `XACQUIRE` prefix in front of the instruction that is used to acquire the lock which is protecting the critical section. The processor treats the indication as a hint to elide the write associated with the lock acquire operation, and a transaction is open instead. If the transaction aborts, the processor will roll back the execution and then resume it non-transactionally. In case of a processor not supporting TSX, the lock is acquired normally, and the execution is serialized. The second interface provided by TSX is known as Restricted Transactional Memory (RTM) and allows more flexibility in transaction declaration than HLE. RTM adds three new instruction to the ISA: `XBEGIN`, `XEND` and `XABORT`. Intel does not provide implementation details of TSX, but gives some hints which suggest that TSX is a *best effort* approach to HTM, like Sun's Rock and AMD's ASF. That is, they do not guarantee successful execution of transactions of any size and duration, and they abort transactions that exceed on-chip resources for HTM, or encounter certain events like page faults, caches misses or interrupts. Thus, Intel enumerates a list of runtime events that may cause transactional execution to abort, namely, synchronous and asynchronous exceptions, memory operations other than write-back cacheable type operations, executing self-modifying code, excessive sizes for transactional regions, conflicting requests to a cache line accessed within a transaction (strong atomicity is ensured), and so on.

2.3. Related Work on Signatures

TM systems must record the address of every memory access issued by transactions in order to detect conflicts between them. These addresses are sorted out into a *read set* (RS) and a *write set* (WS), and they are usually stored in separate structures that are private to each thread context. As conflict detection devices, these structures should not tolerate false negatives (undetected true conflicts) but may assume false positives (false conflicts). In addition, as RS and WS sizes are unknown in advance (unbounded transactions), the number of addresses to be tracked should not be limited. Finally, test and insertion operations should be fast.

Fulfilling the requirements above, Ceze et al. [14] proposed *signatures* as a compact way of representing the read and write sets of transactions by means of Bloom filters, a time and space-efficient hash structure. Since then, signatures have been broadly adopted by several software, hardware and hybrid TM systems to detach conflict detection from caches or accelerate conflict detection. TM proposals that include signatures are FlexTM [97], LogTMSE [110], SigTM [67], STMlite [65], DynTM [59], ... Also, BulkSC [15] proposes a novel way of providing sequential consistency [1] in CMP's that is simple to implement and offers performance comparable to release consistency, by using an underlying HTM-like architecture based on signatures.

Signatures solve certain constraints associated to caches. Modifying caches to track transactional information poses problems on virtualization, since transactions are limited to cache sizes, scheduling time-slice (quantum), migration problems,... Also, cache memories are critical fine-tuned structures that should not be modified by including additional hardware.

Next section describes the Bloom filter structure. Section 2.3.2 deals with hash functions for Bloom filters. Section 2.3.5 discusses different variants of Bloom filters. Section 2.3.3 shows a problem that affect Bloom filters: *the birthday paradox problem.* Finally, Section 2.3.4 describes different signature implementations found in the literature.

2.3.1. The Bloom Filter

The Bloom filter [6] was devised by Burton H. Bloom in 1970 as a time and space-efficient hash coding method with allowable errors. Figure 2.4 shows the design of a Bloom filter. It comprises a bit array of 2^m bits and k different hash functions that map elements into k randomly distributed bits of the array. Such an array is initially set to 0, and inserting an element into the filter consists in setting to 1 the k bits indexed by the hash functions. Test for membership consists in checking that those k bits are asserted. As the array is fixed-sized there exists the possibility of errors of testing, called *false positives.* For instance, in Figure 2.4, elements x, y and z are inserted in the filter and the bits indexed by the hash functions ($k = 2$ in this case) are set to 1. When we test for element w, it happens to be mapped into bits that have already been set to 1, so the test is a false positive. However, false negatives are not possible.

The probability of false positives in regular Bloom filters [10, 92] can be formulated as follows. Consider a Bloom filter that maps a space of N elements, $N = 2^n$, into an array of M bits (indexes), $M = 2^m$, $m \leq n$, through a family of k hash functions, $\{h_0, h_1, ..., h_{k-1}\}$. As each hash function maps one element into one of the M possible bits of the array, and assuming that hash functions yield uniformly distributed indexes (see Section 2.3.2), the probability that one bit is set to 1 in the filter is $\frac{1}{M}$. Hence, the probability that a bit is set to 0 is $1 - \frac{1}{M}$. Consider a sequence of q elements, $\{x_0, x_1, ..., x_{q-1}\}$, to be inserted in the filter.

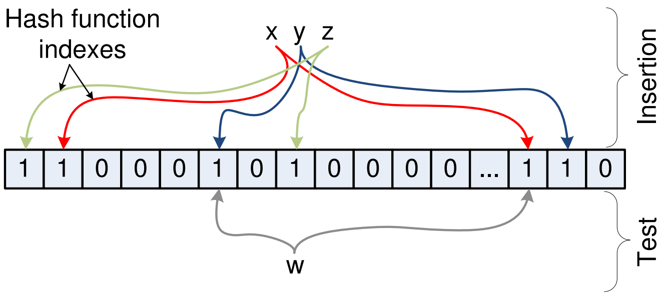


Figure 2.4: Design of a Bloom filter. A false positive scenario.

After the insertion of those q elements using k hash functions per element, the probability that a bit is still 0 is:

$$p_{\text{ZERO}}(M, q, k) = \left(1 - \frac{1}{M}\right)^{qk}, \tag{2.1}$$

assuming that the hash functions are independent each other. The exponent qk can be called the *occupancy* of the filter.

The probability of getting a positive match on testing for membership of an element is:

$$p_{\text{POSITIVE}}(M, q, k) = (1 - p_{\text{ZERO}})^k = \left(1 - \left(1 - \frac{1}{M}\right)^{qk}\right)^k, \tag{2.2}$$

as k bits are checked in each test. A test for membership is a true positive for the q elements inserted in the filter, but not for the remaining $Q - q$ elements that also get a positive match, being Q the number of total positives. So, the probability of getting a false positive is the probability of getting a positive on a test of an element that has not been inserted. According to Bayes' rule:

$$\begin{aligned} p_{\text{FALSE POSITIVE}}(M, q, k) = p_{\text{POSITIVE}}(M, q, k)\frac{Q - q}{Q} \approx \\ \approx p_{\text{POSITIVE}}(M, q, k). \end{aligned} \tag{2.3}$$

This approximation assumes that the number of total positives in the space under test is much larger than the number of inserted addresses, $Q \gg q$, thus, $\frac{Q-q}{Q} \approx 1$. Such an assumption is valid when $N \gg M$.

Expression 2.3 can be simplified by using the Taylor series expansion of the exponential function, $e^x = \sum_{n=0}^{\infty} \frac{1}{n!}x^n$. In our case, since $1/M \ll 1$, the expo-

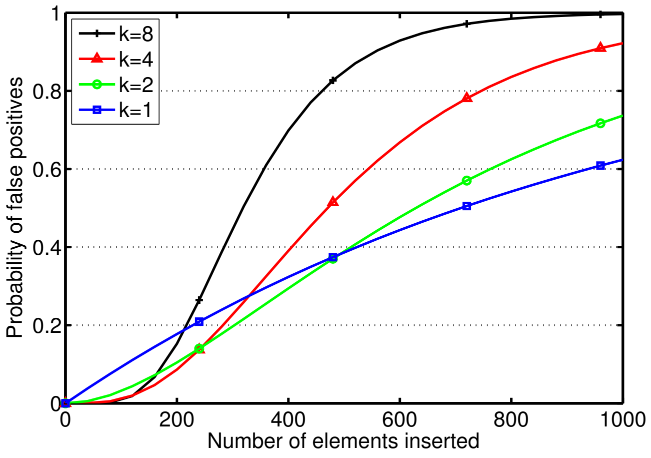


Figure 2.5: False positive probability of regular Bloom filters. $M = 1024$ and $k \in \{1, 2, 4, 8\}$.

nential function can be approximated by the two first terms of the series:

$$e^{-1/M} = 1 - \frac{1}{M} + \frac{1}{2M^2} - \frac{1}{6M^3} + ... \approx 1 - \frac{1}{M},$$

so $p_{\text{ZERO}}(M, q, k) = \left(1 - \frac{1}{M}\right)^{qk} \approx e^{-\frac{qk}{M}}$, and the probability of false positive is:

$$p_{\text{FALSE POSITIVE}}(M, q, k) \approx \left(1 - e^{-\frac{qk}{M}}\right)^k. \tag{2.4}$$

Figure 2.5 shows the probability of false positives given by Expression 2.3, for different values of k and a filter of $M = 1024$ bits. The number of elements inserted in the filter is shown in the x-axis. We can see that better false positive probability is expected for low populated filters and a high number of hash functions ($k \in \{4, 8\}$). However, the more hash functions the Bloom filter has, the earlier the filter populates and the higher the false positive probability is expected for high populated filters. Section 2.3.4 discusses the bibliography references that study the k trade-off in the context of TM.

The false positive rate given by Expression 2.3 is an approximation that assumes the indexes are independent each other and $N \gg M$. Bose et al. [10] proposes an exact formula for the false positive rate in any case, for which they model the problem as a problem on balls and urns. The resulting expression for the exact false positive rate is the following:

$$p_{\text{FALSE POSITIVE}}(M, q, k) = \frac{1}{M^{k(q+1)}} \sum_{i=1}^{M} i^k i! \binom{M}{i} \left\{ {qk \atop i} \right\}, \qquad (2.5)$$

where $\left\{ {qk \atop i} \right\}$ is the Stirling number of the second kind.

Expression 2.3 underestimates the false positive rate in some cases with respect to the exact rate given by Expression 2.5, but we will use the former expression along this thesis as it is very accurate when the assumptions are fulfilled, which is the case, and it is more intuitive and much easier to evaluate.

2.3.2. Hash Functions for Bloom Filters

In the preceding section, we have assumed that hash functions yield uniformly distributed indexes. However, real hash functions are usually biased. Ramakrishna et al. [86] study two different types of hash functions that are commonly used for hardware applications: bit selection and XOR hashing. Figure 2.6 depicts the implementation of each method. With bit selection, the i hash function, $0 \leq i < k$, is made up by extracting the bits i, $i + k$, $i + 2k$,... from the address. On the other hand, in XOR hashing, each hash function consists of an XOR gate tree per hash bit. Bit selection hash functions are more hardware-efficient than XOR ones since they can be implemented by simply hard-wiring the bits corresponding to each hash function. However, they provide less-quality random indexes than XOR hashing functions.

Carter and Wegman [12] presents three classes of hash functions, H1, H2 and H3, whose expected time to map a given sequence of inputs into their corresponding indexes is linear in the length of the sequence. H1 amounts to taking the last bits of the address to make the index, as long as the number of indexes is a power of 2. The class H2 is similar to H3, but the functions require less time and more space, since the address is first mapped into a longer bit string, but one with fewer 1's. The H3 class belongs to the XOR hashing type of hash functions, and it has been proved to exhibit a high quality behavior for memory address streams, close to random distribution [92]. Vandierendonck et al. [104] defines the H3 class of hash functions as a linear transform between an n-bit word and an m-bit word: $h_i : GF(2)^{1\times n} \rightarrow GF(2)^{1\times m}$, being $GF(2)$ the Galois field of two elements, under the bitwise XOR. Then, as H3 functions map addresses linearly

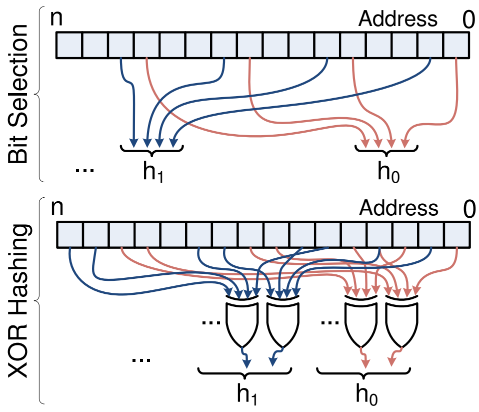


Figure 2.6: Bit selection and XOR hashing function implementation.

into indexes, they can be completely characterized by a matrix in $GF(2)^{n\times m}$:

$$H = \begin{bmatrix} h_{n-1,m-1} & h_{n-1,m-2} & \cdots & h_{n-1,0} \\ h_{n-2,m-1} & h_{n-2,m-2} & \cdots & h_{n-2,0} \\ \vdots & \vdots & & \vdots \\ h_{0,m-1} & h_{0,m-2} & \cdots & h_{0,0} \end{bmatrix}. \tag{2.6}$$

Essentially, it is a $(n \times m)$ binary matrix whose coefficient $h_{i,j}$ is 1 if the bit i of the address is an input bit of the XOR tree which computes the bit j of the index. The hash output $y = h(x) = [y_{m-1}...y_1y_0]$ of an n-bit address with binary expression $x = [x_{n-1}...x_1x_0]$ corresponds to a map $GF(2)^{1\times n} \rightarrow GF(2)^{1\times m}$, which is computed as follows:

$$[y_{m-1}...y_1y_0] = [x_{n-1}...x_1x_0]H. \tag{2.7}$$

For example, a hash function mapping a space of 2^4 addresses into 2^2 possible indexes is:

$$h(x) = [x_3x_2x_1x_0] \begin{bmatrix} 1 & 0 \\ 1 & 1 \\ 0 & 1 \\ 1 & 0 \end{bmatrix} = [x_3 \oplus x_2 \oplus x_0,\ x_2 \oplus x_1].$$

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

2.3.3. The Birthday Paradox Problem: a motivation

The birthday paradox is an unintuitive statistical problem which can be formulated in the following manner: Within a group of about 23 people, you will have a good chance of two people sharing the same birthday. Actually, the exact probability of two people sharing the birthday in a group of 23 people is 50%. By the pigeonhole principle, the probability is 100% within a group of 366 people, as there are 365 days in a year.

The birthday paradox problem can be applied to signatures in TM systems [112] as they use Bloom filters with hash functions that map a set of addresses (people) into a reduced set of bits that comprise the Bloom filter array (days in the year). Then, signature-based TM systems are supposed to encounter conflicts even when transactions update a small set of memory locations. It could be worse if transactions' data sets are large (see Figure 2.5), signatures are small, or if applications spawn many threads.

The birthday paradox problem has motivated most of the works discussed in Section 2.3.4 and the signature enhancements proposed in this thesis.

2.3.4. Signature Implementations

Bloom filters can be implemented as a k-ported SRAM in its regular version. However, Sanchez et al. [92] proposed the *parallel* Bloom filter as an alternative hardware-efficient implementation to regular Bloom filters. Multiported SRAMs require much hardware as they grow quadratically with the number of ports. Figure 2.7 shows the implementation of both regular and parallel filters. Whereas the regular filter is implemented as a k-ported SRAM, the parallel one consists of k subfilters implemented as single-ported SRAMs, yielding the same or better false positive rate. The probability of a bit of a particular subfilter being 0 after q insertions is:

$$p_{\text{ZERO}}(M,q,k) = \left(1 - \frac{1}{M/k}\right)^{q}, \tag{2.8}$$

following the same notation and assumptions of Section 2.3.1. Thus, the probability of getting a false positive in a parallel Bloom filter is:

$$\begin{aligned} p_{\text{FALSE POSITIVE}}(M,q,k) \approx p_{\text{POSITIVE}}(M,q,k) = \\ (1 - p_{\text{ZERO}})^{k} = \left(1 - \left(1 - \frac{1}{M/k}\right)^{q}\right)^{k}, \end{aligned} \tag{2.9}$$

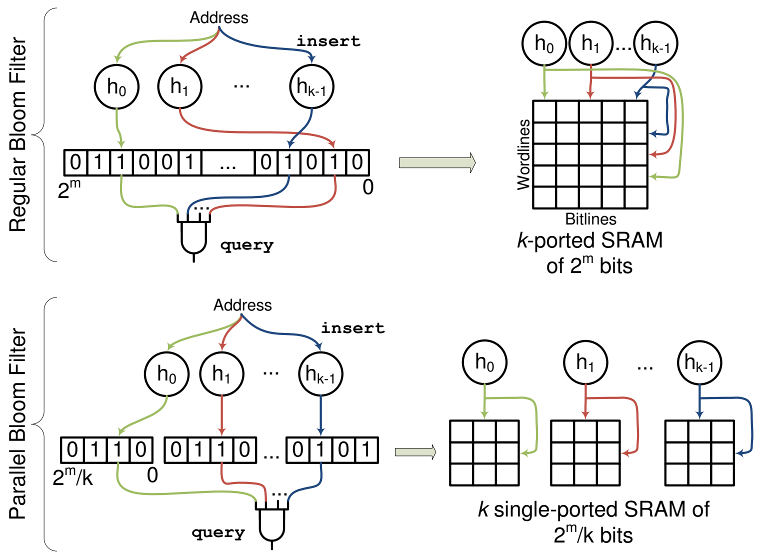


Figure 2.7: Regular Bloom filter vs. Parallel Bloom filter. Design and implementation.

which appears to be different from the Expression 2.2 for regular Bloom filters. However, by the Taylor series approximation of e^x, we get that, if $k/M \ll 1$, then $1 - \frac{1}{M/k} \approx e^{-k/M}$, so:

$$p_{\text{FALSE POSITIVE}}(M, q, k) \approx \left(1 - e^{-\frac{qk}{M}}\right)^k, \tag{2.10}$$

which is the same false positive rate as the true Bloom signature, shown by Expression 2.4, as long as $k/M \ll 1$, the normal case. In the same work, Sanchez et al. found that H3 hash functions perform better than bit-selection, and four or more such hash functions should be used.

Cuckoo-Bloom signatures are also proposed in [92]. They are intended to perform like high-k Bloom filters for small transactions, while yielding the false positive rate of Bloom filters with few hash functions when transactions are large. Cuckoo-Bloom filters act like a hash table in the beginning of the transaction. Addresses are stored as if in a set-associative cache, where tags and data are the result of hashing the address with two independent hash functions, and sets are indexed by other hash function. When a set is full, the filter executes a sequence of evictions and re-insertions to store the incoming address. If such a sequence takes too long, the set is converted into a regular Bloom filter with low

k. The addresses in the set are previously stored in a separate storage space for eventually being inserted into the newly converted Bloom filter. Lookups are fast, but insertions are more complicate, and the filter needs certain control logic, additional storage, a bit array to signal whether a set has been converted into a Bloom filter or not, and other structures (comparators,...) that complicate the design and might rise the hardware budget.

An alternative hardware-efficient implementation of hash functions, Page-Block-XOR hashing (PBX), is proposed in [111]. They use the concept of entropy to find the highest randomness bits of the address, to allow reducing the hardware complexity of hash functions. Notary [111] also proposes a technique to reduce the number of asserted bits in the signature. Their approach is based on segregating addresses into private and shared sets. Then, only the shared addresses are recorded. This solution requires support at the compiler, runtime/library and operating system levels. In addition, the programmer must define which objects are private or shared, which might be a difficult and error-prone task.

Titos et al. [101] propose a directory-based scheme for detection of conflicts in HTM. They detach conflict detection from the L1 caches and shift it to the directory level. This approach optimises eager conflict detection HTM systems with an unordered and scalable network, when running applications with high number of conflicts. The network traffic is reduced up to 30% since the directory does not have to send signature check messages to the cores. Furthermore, by having the signatures of each core centralised into the directory, they can perform more efficiently, since transactions usually access the same shared data, which is not kept redundantly into the directory.

Orosa et al. [75] propose *FlexSig* as a flexible hardware signature implementation to change dynamically the amount of signatures per core according to system requirements. FlexSig groups all signatures in the system into a pool of signatures and assigns them to the cores on demand. It relies in the fact that all cores are not always running transactional code at the same time. Thus, if there are only two transactions running in the system, they will use half of the signature pool each. If other cores start a transaction, they demand signature allocation to the pool and it is repartitioned to meet the necessities of all the cores running transactions in the system, without incurring false positives.

Choi and Draper [19] propose adaptive grain signatures, that keep the history of transaction aborts and dynamically changes the input bit range to the hash functions on the abort history. The aim of this design is to reduce the number of false positives that harm the execution performance.

Recently, Choi et al. [20, 21] have proposed a unified signature design that merges read and write signatures into a single one. Two variants of unified signatures are shown. One which tracks transactional accesses regardless of whether they are reads or writes, and another which includes a helper signature to filter out read-read conflicts. Chapter 6 proposes signatures schemes that also consider the basic unification of read and write filters, but this baseline design is optimized by a different strategy, based on partially keeping separate read and write hash functions that operate on the same subfilter. Both works are complementary.

2.3.5. Bloom Filter Variants

Several Bloom filter variants have been proposed in the literature in order to adapt generic Bloom filters to different contexts of application. Next, we show related Bloom filter proposals to that we propose in this thesis.

Distance-sensitive Bloom filters [51] use *locality-sensitive* hashing [16, 47] to formulate queries of similarity in metric spaces using compact representations of objects. That is, instead of testing for membership as in a generic Bloom filter, we can query whether an element is close to other element in a set. Locality-sensitive hashing is used, since we can define hash functions such that the hash values of two elements that are close each other can match with a given probability. The distance-sensitive Bloom filter defines a family of locality-sensitive hash functions, $\{h : N \rightarrow M\}$, that are (r_1, r_2, p_1, p_2)-sensitive with respect to a metric space (N, d), where d is the distance metric over the set N, if $r_1 < r_2$, $p_1 > p_2$, and for any $x, y \in N$,

- if $d(x, y) \leq r_1$ then $Pr(h(x) = h(y)) \geq p_1$, and
- if $d(x, y) > r_2$ then $Pr(h(x) = h(y)) \leq p_2$.

With such a family of hash functions, the distance-sensitive Bloom filter can provide a fast answer without comparing against the entire set, M, or without performing a full nearest-neighbor query, which can be very useful in the context of distributed databases, networking and even DNA sequencing. However, the filter can produce false positives when x is far away from y and the filter response is positive, and false negatives when x is close to y and the filter answers that it is far away.

Hao et al. propose *combinatorial* Bloom filters [38] as a fast dynamic multiset membership testing data structure. They use a single Bloom filter and multiple sets of hash functions to map elements depending on the set (group) identifier. The group identifier points out which sets of hash functions have to be used to map the elements into that group. For instance, let 5 be the number of hash function sets, and we want to insert an element x of the group number 7. Such a group has assigned the sets of hash functions 1, 3 and 4. Then, x is hashed using such sets of hash functions and the corresponding bits of the Bloom filter are asserted. In order to determine the group of an element y, such an element is hashed using the 5 sets of hash functions. If sets 1, 3 and 4 result in a 1, element y can be identified as being a member of group number 7. Other group identifiers use different sets of hash functions. Combinatorial Bloom filters need many filter checks especially when there are large number of sets.

Fan et al. [31] propose a *counting* Bloom filter that allows not only insertion and test for membership operations, but also deletion of an element from the filter. This is done by maintaining a counter per bit of the array. All the counters are initially set to 0, and whenever an element is inserted or deleted, the counters indexed by the hash functions are incremented or decremented accordingly. A test for membership checks that each counter indexed by the hash functions is non-zero. A false negative can arise if a counter overflows, as subsequent deletions can set the counter to 0 and the overflowing insertions are not taken into account. To keep the filter from yielding false negatives, the counter can be turned off on overflow so that deletions are not allowed on that counter. On the other hand, the *spectral* Bloom filter [25] uses counters to allow the filtering of elements with multiplicities.

The *Bloomier* filter [18] propose to associate a value to each element inserted in the filter, thus transforming a set of keys into a map of pairs (key, value). The simplest form of a Bloomier filter is that of having a map that only permits two values, e.g. 0 and 1. Thus, the Bloomier filter has two Bloom filters, one that tracks the keys whose associated value is 0, and another for the keys whose value is 1. On a search for a given key x, both filters are probed. If only the first filter is a hit, x has a high probability of having associated the value 0, and vice-versa. If both filters happen to have the key, this look up procedure has to be recursively repeated on a set of pairs of smaller filters that contain the keys that produced a false positive in the preceding stage. On inserting a key y with value 0, we insert y in the first filter and check if the second filter also contains y. If so, y has to be inserted in the next pair of smaller filters, and so forth. In order to allow general values other than 0 and 1, we can have one simple Bloomier filter per bit of the value.

3 Methodology

This chapter deals with the methodology we followed to evaluate our proposals. The simulation environment we have used for experimentation is described in Section 3.1, and the benchmark suite is reviewed in Section 3.2.

3.1. Simulation Framework

For evaluating the proposals in this thesis we have used a full system execution-driven simulator called Simics [60], which is introduced in Section 3.1.1, along with the HTM module for Simics called GEMS [63]. GEMS is provided by the Wisconsin Multifacet Project as open-source and it is described in Section 3.1.2. Section 3.1.3 deals with the target CMP system organization and Section 3.1.4 outlines the TMtool, a straightforward functional TM simulator that we have built using Intel's PIN [57].

3.1.1. Simics Simulator

Simics is a system level instruction set simulator. Unlike emulators that are focused on executing a program, Simics was designed not only to run programs, but also to gather runtime information, and to allow user-developed add-ons. Simics is a system level simulator since it can boot unmodified operating system kernels.

Simics can run on different *host* machines on top of several *host* operating systems. In our case, we have run Simics 2.2.19 on SUSE Linux 10.1 over an

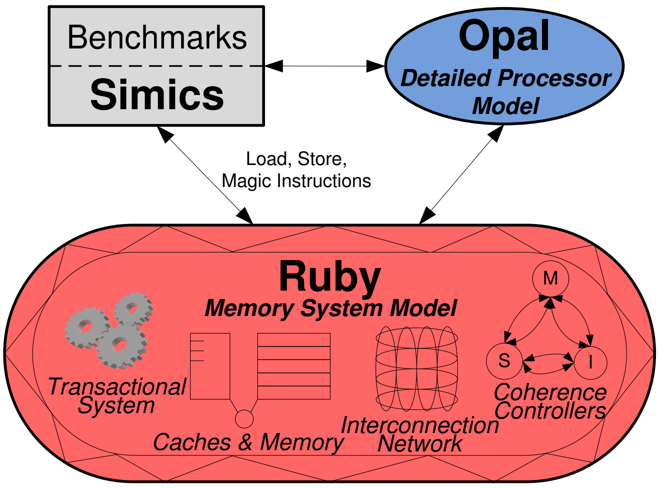


Figure 3.1: Simics+GEMS architecture: the memory module, Ruby, can be driven by either Simics or Opal's out-of-order processor module.

8-core AMD64 architecture. As regards the *target* system, Simics implements a list of processors and servers from ARM to Intel amongst others, regardless of the host type. We configured Simics to target a *Serengeti* system that simulates the Sun Fire 6800 server with UltraSPARC-III Cu processors, since GEMS module is tied to the ISA of such processors. We installed Solaris 10 on the simulated machine with the development tools for compiling the benchmarks.

Figure 3.1 shows the roll of Simics within the simulation framework. Simics provides an *in-order* model by default, where benchmarks run on top of the OS and Simics functionally simulates the instruction set and executes program instructions sequentially. This simulation model is fast and simple but it fails to provide a detailed execution timing model. However, Simics' in-order model can be extended by adding *timing models* to control the timing of memory operations and processor instructions. Simics provides interfaces by which a processor can stall while a memory operation (an instruction fetch or a data transaction) is being simulated by a detailed memory timing model. Likewise, Simics provides the user with a micro-architectural interface for implementing out-of-order processors. In *out-of-order* mode, the user's processor timing model tells Simics whenever an instruction can be committed. By using these features, Wisconsin Multifacet group developed a multiprocessor detailed timing toolset for Simics called GEMS, which is described in next section.

3.1.2. GEMS Toolset

The Wisconsin Multifacet Project's *General Execution-driven Multiprocessor Simulator* (GEMS) is a toolset for Simics released under GNU GPL [33]. GEMS decouples simulation functionality and timing by leveraging Simics functional simulator with two main modules (see Figure 3.1):

1. *Opal*: This module models a SPARC v9 processor with dynamic scheduling. Opal uses the micro-architectural interface provided by Simics for filling the pipeline with instructions. Then it fetches, decodes, dynamically schedules, and executes such instructions, and it is also capable of predicting branches and speculatively accessing the memory hierarchy. Once Opal has an instruction to retire, Simics processor is instructed to advance one instruction and Opal compares its processor state with that of Simics to ensure a correct execution.

2. *Ruby*: Ruby is a timing simulator of a multiprocessor memory system. Ruby models caches, cache controllers, system interconnect, memory controllers, banks of main memory as well as the LogTM-SE HTM system [110]. Ruby implements cache coherence independent components like the interconnection network, cache arrays, memory arrays, message buffers, and glue logic so that they are assigned a hard-coded timing. However, the time count is eventually determined by how the coherence protocol manages fetch and data transactions coming from the processors. For specifying the memory coherence protocol, Ruby provides a domain-specific language called SLICC (Specification Language for Implementing Cache Coherence). With SLICC one is able to specify cache and directory controller state machines in terms of states, events, transitions, and actions over memory blocks.

Throughout this thesis, we used the Ruby module driven by Simics' default in-order processors. Whereas running Simics with Ruby slows down simulations by about 125×, adding Opal's processor model worsens the slowdown to about 300×, so Opal was discarded for simulations. With the in-order driver, Ruby receives a Simics' request (load, store, or instruction fetch) which is checked for hitting the first level cache. If so, Simics is not stalled by Ruby, thus retiring the instruction and switching to the next processor, as far as a multiprocessor setting is concerned. If the first level cache request is a miss, Ruby stalls the processor issuing the request and simulates the cache miss. Each processor can have only a single miss pending, but the memory model determines the stalling time for each processor which in turn makes processors overtake each other, thus simulating the memory request interleaving of real multiprocessors.

Moreover, the Ruby module adds pseudorandom delays to the latency of memory accesses to deal with variability in simulation experiments. Variability is a well-know phenomenon in real systems that is often ignored in simulations. Therefore, multiple runs of each experiment were conducted to obtain confident error bars [2] with a confidence probability greater than 95%.

Finally, Simics and Ruby maintain a communication link via Simics' magic instructions. Magic instructions are special no-operation instructions (`sethi n,%g0` in case of SPARC processors) that can be used within benchmarks running on the target system to pass information to Simics modules. In this case, magic instructions' main role is to place the boundaries of transactions in the benchmarks so that Ruby's transactional system knows when they start and end. Magic instructions are also used for signaling when the Ruby module must be loaded and when it must be unloaded. In a normal simulation with Simics and Ruby, the benchmark initializes all data structures and spawns the threads with the Ruby module unloaded to speed up the simulation. Then, before entering the parallel phase of the benchmark, the magic instruction is executed and Ruby is loaded to profile the parallel execution. Once the parallel phase is done, Ruby is unloaded an the simulation is over. The sequential phase and the spawning of the threads are not simulated in detail, thus saving time.

3.1.3. Target System

We have used the same target system configuration for all experiments in this thesis. Table 3.1 shows the parameters for the modeled system. The base CMP system consists of 16 in-order, single-issue cores. Each core has a 32KB split private L1 cache and one bank of a 8MB unified shared L2 cache, comprising a tile from the tiled organization shown in Figure 3.2. A packet-switched interconnect connects the cache banks in the tiled topology consisting of 4 clusters, each made up of 4 cores, following a NUCA (Non-Uniform Cache Access [50]) scheme where the access to non-local L2 cache banks is slower than that for the local cache bank. Latencies for caches, memory and network are shown in Table 3.1 although hit/miss latency is eventually determined by the coherence protocol and the network hops of each message needed to resolve the memory transaction. Cache coherence implements the MESI protocol and maintains an on-chip directory in L2 cache which holds a bit vector of sharers. Inclusion property is imposed in the cache hierarchy to reduce the cache coherence complexity [4].

Table 3.1: Base CMP system parameters.

Processor	16 in-order single-issue cores
	IPC=1 for non-memory operations
L1 cache	Private
	Split, 32KB instructions + 32KB data
	4-way set-associative, 64B blocks
	1 cycle hit latency
L2 cache	Shared banked
	Unified, 8MB (16 banks of 512KB)
	8-way set-associatives, 64B blocks
	6/20 cycle tag/data access latency
L2 directory	Full bit-vector of sharers
	6 cycle access latency
Main Memory	8GB
	450 cycle access latency
Network	Packet-switched tiled interconnect
	1 cycle link latency

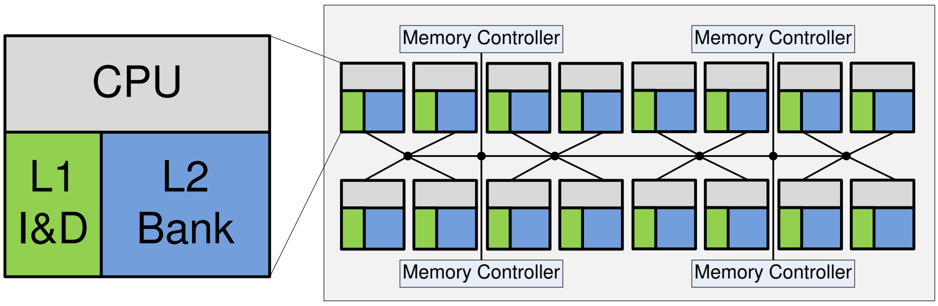


Figure 3.2: Simulated CMP system organization.

As regards the TM system, we have used the baseline LogTM-SE HTM [110]. LogTM-SE leverages each thread context state of the CMP with the structures shown in Figure 3.3 (green blocks marked with dashed lines). A register checkpoint and the program counter for the beginning of the transaction are needed on abort to retry the transaction. Several registers are needed to store the log base address, the log head pointer, the depth of current transaction nesting level and the address of the software handler called on abort. LogTM-SE also adds a log filter that holds recently logged blocks to prevent the system from writing the same block to the log more than once within a transaction. Finally, two signatures are added, one to store the RS and the WS of the running thread's transaction and another, the summary signature, to track conflicts with suspended thread's

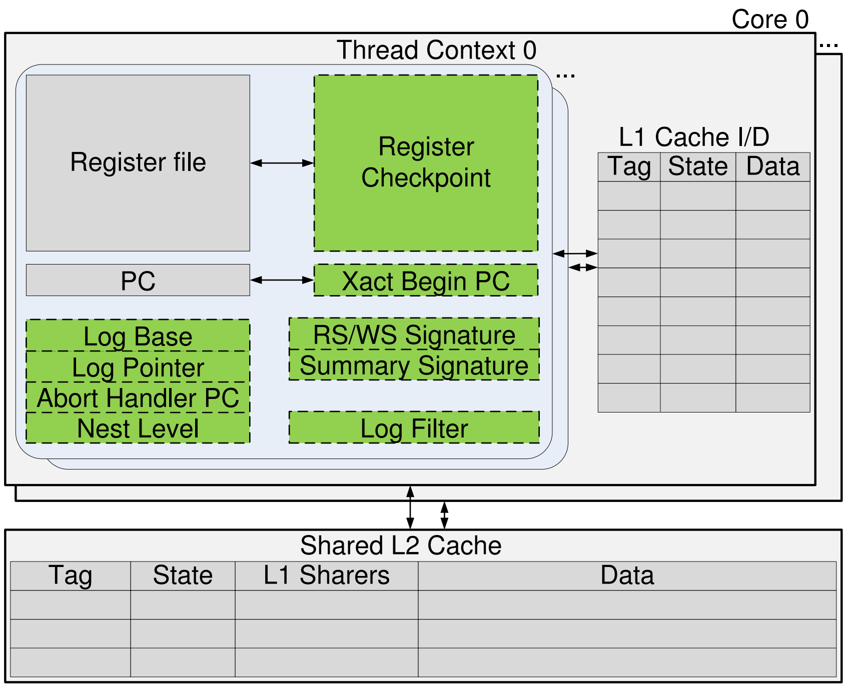


Figure 3.3: LogTM-SE hardware add-ons.

transactions. Caches are left untouched.

In our target system, cores are single-threaded and threads are bounded to the cores (see Section 3.2.2) so summary signatures are not used. The default eager conflict detection and eager version management schemes are used, with the base conflict detection policy where requester stalls, retries after a backoff time, and aborts on a possible deadlock cycle. Conflict detection is performed on cache block granularity with signatures operating on physical addresses. Perfect signatures keep track of every single address read or written in a transaction without yielding false positives. They are hardware unimplementable but we use them as a reference in our simulations.

3.1.4. TMtool: A Simple Functional TM System

Prior to include some signature proposals into the cycle accurate target HTM simulator, we developed a straightforward functional eager-eager TM system to gather preliminary results and benchmarks characteristics. Intel's PIN instrumentation tool [57] was used to quickly implement the system. Pin is a tool for the dynamic instrumentation of applications and supports Linux, Windows and MacOS binary executables for a wide range of Intel® processor architectures. Pin provides functionality to inject arbitrary code (written in C or C++) at arbitrary places in the executable. Unlike other tools, Pin adds code dynamically into running applications instead of statically instrumenting executables by rewriting them.

Two components can be differentiated when instrumenting an application with PIN: (i) instrumentation code, which decides where the analysis code is inserted and what analysis code is to be added; and (ii) analysis code, which is the code to execute at insertion points. PIN intercepts the execution of the first instruction of a program and instruments it dynamically. In our case, the instrumentation code is implemented in two different ways:

1. *Alarms to detect* `xact_begin` *and* `xact_end`: Transaction beginnings and endings are detected via PIN's alarms. The executable to be instrumented must have transactions enclosed by calls to `xact_begin()` and `xact_end()`, defined as empty functions. The application should call such functions before the beginning and after the end of a critical section. Whenever the instrumentation alarm finds a call to `xact_begin`, the analysis function for opening a transaction is inserted. We implemented the analysis function so that a new transaction is allocated with its log, signatures and statistics. The thread context is checkpointed in order to rollback in case of abort. The analysis function also signals subsequent instructions as transactional by setting an `insideTransaction` flag. In case of `xact_end()`, the inserted analysis function is for ending the transaction, and it resets the `insideTransaction` flag signaling that subsequent instructions are not transactional anymore. Transactional statistics are also dumped into a file. As it is an eager-eager system, log and signatures are discarded.

2. *Instruction instrumentation mode*: PIN offers the instruction instrumentation mode which lets the tool inspect and instrument an executable a single instruction at a time. We added an instrumentation function in this mode and it is essentially an `if-then` statement in which we check if the inspected instruction is either a memory read or a memory write and if the `insideTransaction` flag is set. If the condition is true, then the analysis function for registering the memory instruction is inserted before the instruction. If it evaluates to false, then nothing is done. In case of a memory read, the analysis function checks for a conflict with signatures of other

transactions running in the system. If no conflict is detected, the location is registered in the local signature. Otherwise, a conflict is signaled and the local transaction is aborted by undoing the log and rolling back to the checkpoint. A memory write is similar to a memory read but it stores in the log the old value of the memory location to be written.

Our PIN TM simulator implements a naive conflict resolution policy that aborts every transaction detecting a conflict. More complex policies supporting transaction stall and back-off waiting like LogTM's could have been implemented but it implied logically ordering transactions with timestamps and implementing a conflict handler to detect potential deadlocks. As the system was intended for gathering some preliminary results and benchmarks characteristics, we decided to leave it for future work.

Finally, our PIN TMtool shows statistics of the number of committed transactions, the number of aborted transactions, the size of read and write sets of transactions and statistics of the occupancy in the filters. We can get traces of the memory footprint of given transactions and it is easy to modify for getting other transactional numbers. Notice that certain statistics, like the number of aborts, might depend on the interaction of threads so an application with several running threads could yield different statistics on a single-core machine than on a CMP. Simics, though, is platform independent.

3.2. Benchmark Suite

For the evalulation of the proposals presented in this thesis we used all the benchmarks belonging to the Stanford's STAMP suite [66]: *Bayes, Genome, Intruder, Kmeans, Labyrinth, SSCA2, Vacation* and *Yada*. This suite is designed for Transactional Memory research and includes a wide range of applications laying emphasis on those with long-running transactions and large read and write sets. Such benchmarks are of special interest for signature evaluation because they put the most pressure in signatures.

3.2.1. Benchmark Description and Characterization

STAMP applications cover a wide range of computing domains and transactional characteristics. Next, we provide a brief description of the eight benchmarks comprising the suite:

1. *Bayes*: This application implements an algorithm for learning Bayesian networks which are an important part in the machine learning domain. Learning a Bayesian network consists in maximizing a likelihood score function which evaluates the *goodness* of the Bayesian network. Such maximization is carried out by using a hill-climbing technique where an incremental change is made to the Bayesian network iteratively until no further improvements can be found. The algorithm implements the network as a directed acyclic graph (DAG) with random variables as nodes and their conditional dependences as edges. The conditional dependences are added to the network by analyzing the observed data.

   The critical sections that calculate and add new dependencies to the network are enclosed by transactions. The lock-based version of this application is not as simple as the transactional approach. A two-phase locking method with deadlock detection is needed for modifications in the graph.

2. *Genome*: This application is located in the domain of bioinformatics. *Genome* implements a gene sequencing program that reconstructs the gene sequence from segments of a molecule of DNA which may have been extracted with a DNA sequencing instrument, or DNA sequencer, in terms of nucleotide bases: adenine, guanine, cytosine, and thymine (AGCT). The DNA sequencer can yield many duplicates of gene segments. Therefore, the first step of the algorithm is deleting duplicate segments for which it is used a hash set that stores unique elements only. The second step is matching segments using the Rabin-Karp string search algorithm. Rabin-Karp's algorithm speeds up the search of substrings in a string by using a hash function. Each thread is constructing its own gene partition of unmatched segments from a pool. The final step is building the entire sequence and it is not parallelized.

   Transactions are used in the first step of the algorithm to ensure concurrent access to the hash set. In the second step, transactions are used in the access of to the global pool of unmatched segments as threads may try to remove the same segment. Deadlock avoidance has not to be implemented by using transactions.

3. *Intruder*: This application is located in the domain of security and implements a network intrusion detection system (NIDS) based on signatures. Network packets are scanned to search for matches against a known set of intrusions. This search is performed in parallel through three stages: (i) *capture*, where a packet fragment is dequeued from a FIFO queue; (ii) *reassembly*, where the fragments belonging to the same session are stored in

a self-balancing tree; and (iii) *detection*, where the reassembled packet is checked against the data base of signatures of known attacks.

The complexity of the reassembly stage makes the implementation with fine-grain locks a fairly difficult programming task, to such an extent that the algorithm is usually implemented with coarse-grain synchronization. The transactional version also uses coarse-grain transactions but allowing optimistic concurrency.

4. *Kmeans*: This application is located in the interdisciplinary field of data mining and it is an example of unsupervised machine learning. The K-means algorithm is used to separate a set of objects into K clusters or groups with similar characteristics. Each thread is assigned a partition of the object space and each works iteratively on its partition. Updates to the center of the clusters are protected with a transaction since different threads might be working on the same center. The larger the K the lesser probability of conflict between threads. Consequently, *Kmeans* can benefit from optimistic concurrency of transactions.

5. *Labyrinth*: This application is located in the domain of engineering and it is an implementation of Lee's algorithm to calculate the paths between a set of starting and ending point pairs in a three-dimensional grid. It can solve problems inherent in wiring diagramming, and optimal route finding. Each thread takes a pair of points and calculates the path between them. A conflict occurs when paths from different threads overlap in some point. Each thread first privatizes the grid at the beginning of the calculation of the path to reduce conflicts. At the end of the calculation, the thread must check the global grid for overlaps with the new path, since it might have been working with a stale grid due to path additions by other threads.

   The transactional version of *Labyrinth* does not need the implementation of deadlock avoidance techniques required by the lock-based approach. However, to improve the performance attained by transactions, an early-release [44, 98] technique is used to delete the locations read in the privatization of the grid from the transaction's signature to reduce conflicts. As early-release is not implemented in our HTM baseline system we used open nested transactions to enclose the privatization of the grid. Thus, once the grid has been read by a thread without any conflict, the open nested transaction is committed releasing isolation of the global grid.

6. *SSCA2*: This application implements four graph kernels that are frequently used in the scientific domain ranging from computational biology to security. *SSCA2* stands for Scalable Synthetic Compact Applications 2 and it

is based on large, directed, weighted multi-graphs, graphs that allow multiple or parallel edges between two nodes. The suite implements Kernel 1 which is the graph construction. The algorithm parallelizes the insertion of nodes into the graph and it uses adjacency arrays that must be protected to ensure correctness when accessed concurrently.

The transactional version of this application protects the access to the adjacency arrays with transactions. This kind of operation can benefit from optimistic concurrency.

7. *Vacation*: This application is located in the domain of on-line transaction processing and implements a client-server program of a travel reservation system based on a non-distributed database. The database consists of four interrelated tables implemented as Red-Black trees which are self-balancing binary trees. *Vacation* consists of several client threads interacting with the database via the system's transaction manager. Three types of interactions are allowed: reservations, cancellations, and updates. Each of these client interactions comprises a critical section as it modifies the database.

   In this case, using transactions simplifies the parallelization of the code. Implementing an efficient lock-based version of *Vacation* can be complicated due to all data structures involved.

8. *Yada*: Standing for Yet Another Delaunay Application, this application implements the Ruppert's algorithm for Delaunay mesh refinement and it is located in the scientific domain. The algorithm starts from a queue of triangles to refine. Each thread picks a triangle from the queue and executes the refinement. Threads refine the triangle depending on both the surrounding triangles in the mesh of triangles (implemented as a graph), and the mesh boundary segments (implemented as a set). The refinement might yield new triangles to be refined, so they are inserted in the queue for idle threads. The algorithm continues until the queue is empty.

   Transactions make it simpler to parallelize the code of *Yada*. One transaction encloses the extraction of triangles from the queue. The whole retriangulation is enclosed by a transaction, and this part, which concurrently modifies the shared graph, is specially difficult to program under a thread-based model.

Although a characterization of the benchmark suite depends on the given workload setup, we can find in Table 3.2 a qualitative description of the benchmarks in terms of transaction length, read set and write set sizes, time spent inside transactions and amount of contention (probability of conflict). We can

Table 3.2: STAMP benchmark suite qualitative characterization.

Benchmark	**Xact Length**	**RS/WS Size**	**Time in Xact**	**Contention**
Bayes	Long	Large	High	High
Genome	Medium	Medium	High	Low
Intruder	Short	Medium	Medium	High
Kmeans	Short	Small	Low	Low
Labyrinth	Long	Large	High	High
SSCA2	Short	Small	Low	Low
Vacation	Medium	Medium	High	Medium
Yada	Long	Large	High	Medium

see that STAMP provides us with a wide range of distinct characteristic benchmarks. In the suite prevails medium to large transactional set benchmarks which fits our interest in signatures. Finally, benchmarks like *Kmeans* which usually exhibits small RS/WS sizes can be changed to show higher data set sizes by means of the input parameters. This way, we will show input parameters and characteristics, amongst other useful data, in each chapter for quantitative workload characterization.

3.2.2. Benchmark Adaptation to the Simulation TM System Environment

STAMP benchmarks were adapted to the simulation TM system environment, i.e. Wisconsin's LogTM-SE implemented in GEMS, by introducing the following changes to avoid certain undesirable interferences or interactions:

1. *Avoiding OS interference*: We used the pset_bind system call from Solaris OS to bind each thread of the application to a logical processor set. The psrset utility was used to create the logical processor sets in the simulated machine so that one real processor is assigned to one logical processor set. Binding threads to processors keeps the operating system from descheduling and migrating such threads. Surviving to scheduling quantum and migration are events supported by LogTM-SE virtualization but we are interested in signature optimization, so we bound the threads to suppress sources of variability from OS interference. Also, for our simulations, we used 15 out of 16 cores in the simulated system. The remaining processor is left to the OS so that it does not interrupt simulations. Finally, other OS interferences were eliminated. We traversed the benchmark memory footprint before starting computation to avoid page faults inside transac-

tions. Dynamic library functions used inside transactions were also called before entering transactions to let the linker fill in the Procedure Linkage Table (PLT). This way, we disable OS's dynamic linker interference inside transactions since the PLT has all the information about dynamic linking procedures.

2. *Avoiding system library calls*: Some system library calls like malloc or random use global data structures that can lead to undesirable and unjustified conflicts between threads when used within transactions. For example, in each call to random a global variable that holds the last generated random value is read and then written with the new generated random value. This could lead to transaction serialization. On the other hand, malloc invokes system calls operated by the OS in kernel mode that cannot be stalled or aborted by user-level transactions. This might yield isolation consistency errors as non-transactional kernel code from the system call may access global data structures modified by other threads within transactions. Therefore, to avoid system library calls within transactions it was used a per-thread memory pool instead of malloc to allocate dynamic memory. For random, a Mersenne twister pseudorandom generator [64] was used. The implementation of the aforementioned libraries and others can be found in the standard version of STAMP although some modifications were done.

3. *Padding to get rid of false sharing*: Shared data structures were padded to avoid false sharing at cache line level.

4. *Other changes*: Some transactions in *Vacation* benchmark were split to improve scalability for small signatures. In *Labyrinth*, the code that privatizes the grid was enclosed in an open transaction to avoid inserting in the signature those reads (see Section 3.2.1).

4 Locality-Aware Interval Filters

Bloom filters has been used in HTM systems since Ceze et al. [14] first proposed to used them in signatures. Elements inserted into the Bloom filter correspond to memory address locations issued by a running program. However, a Bloom filter may give rise to false positives that can seriously harm the performance, specialy when transactions are long-running and large. Also, Bloom filters do not take into account the fact that memory accesses usually exhibit the locality of reference property.

In this chapter, we contribute with the design and analysis of a hardware alternative to Bloom filters that has been called the *Interval Filter* (IF) [81]. Compared to a classical Bloom filter, the Interval Filter may show a lower false positive rate for those inserted elements that exhibit spatial locality according to a metric space.

Hereinafter HTM is adopted in order to evaluate the proposed filter. However, results could extrapolate to other similar domains where Bloom filters are already used, like a wide range of applications in the domain of networks [11] and file searching [49]. Locality of reference will be exploited to store the locations read and written in an alternative way to Bloom filters, aiming to reduce false conflicts and enhance the execution of large transactions.

The rest of the chapter is organized as follows: Section 4.1 defines the filter and explains how it operates. Section 4.2 describes the simulation environment and evaluates the filter.

4.1. The Interval Filter

The Interval Filter (IF) is proposed to reduce false positives in the presence of locality according to some metric. Without loss of generality, in the rest of the chapter memory addresses will be considered as the elements to be inserted in the filter. Thus, intervals are defined as chunks of consecutive addresses that can be extracted out of a memory reference trace. Figure 4.1 shows the design of the filter. The IF comprises n intervals that are recorded as a pair of two full addresses, one representing the lower bound of the interval and the other one representing the upper bound. A valid bit per interval is also needed, $V_0, ..., V_{n-1}$. Each interval bound has two bit lines. Lower bounds are compared with the incoming address incremented by one and upper bounds are compared with the address decremented by one. Hence, $=^l_0, ..., =^l_{n-1}$ return true if the incremented address is equal to the corresponding lower bound of the interval. On the other hand, $>_0, ..., >_{n-1}$ return true if the address is greater than the lower bound. Likewise, $=^u_0, ..., =^u_{n-1}$ and $<_0, ..., <_{n-1}$ are the bit lines for the upper bounds of the intervals. The filter can be thought of as an extended full-associative cache.

Same primitive operations than Bloom filters can be performed with the interval filter. Figure 4.1 shows, within dash-line boxes, how test for membership and insertions can be implemented. Test for membership consists in checking the *Match* line to be true. This output line is computed by checking that the incoming address is within an interval. To do so, $>_0, ..., >_n$ and $<_0, ..., <_n$ bit lines can be used in the way shown in Figure 4.1. Thus, lookups are relatively fast but insertions are slower and more complicate, as in Cuckoo-Bloom filters [92]. Actually, three cases come up on inserting an address into the interval filter, given that the address is not a member yet. Figure 4.2 depicts the insertion algorithm flow chart:

Case 1 If every $=^l_0, ..., =^l_{n-1}$ and $=^u_0, ..., =^u_{n-1}$ bit lines are zero it means that none of the valid intervals can be expanded, so the incoming address must form a new interval in the filter. Thus, if the filter is not *Full* then the address is inserted into a non-valid interval by storing the original address (neither incremented nor decremented) in both bounds, lower and upper. Conversely, if the filter is *Full* then a valid interval is widen introducing false positives. In order to minimize the number of false positives due to widening, the closest interval bound is chosen. To do so, the address is XORed with the bounds whose $>$ or $<$ bit lines are set to 0. Then, the lower one is chosen as the candidate to store the address in an iterative manner.

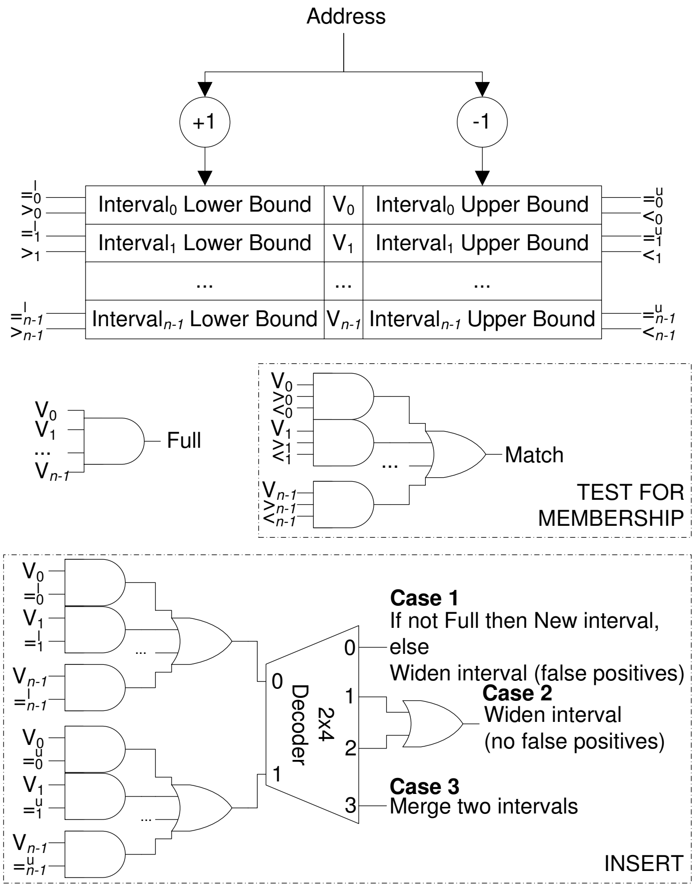
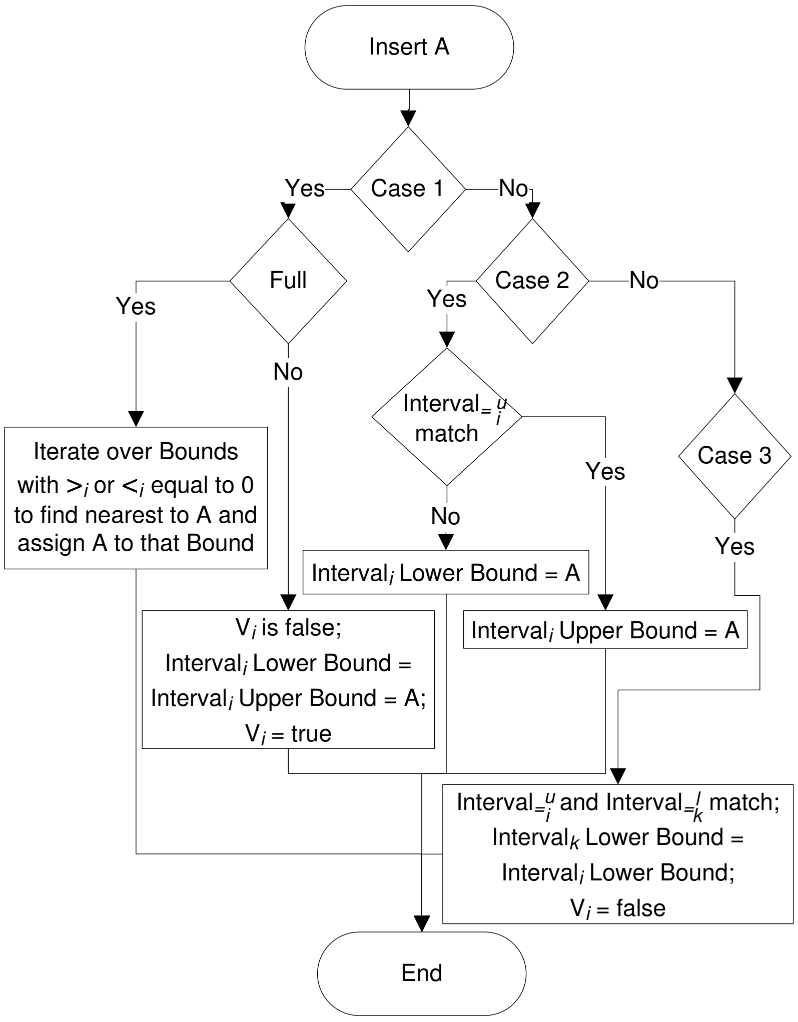


Figure 4.1: Interval filter design.

Case 2 If either only one lower bound or only one upper bound is equal to the incremented/decremented address then an existing interval is to be widen. This can be done in a straightforward way by only storing the original address into the matched bound.

Case 3 If one lower bound and one upper bound are matched at the same time it means that the incoming address is the only one missing to merge two existing intervals. Therefore, one of the two matched intervals is invalidated by clearing its V bit and the remaining interval is widen by setting its lower/upper bound to the lower/upper bound of the invalidated interval. In Figure 4.2 the invalidated interval is i and it has been matched in the


Insert A
Case 1
Yes
No
Full
Yes
No
Case 2
Yes
No
Iterate over Bounds with $>_i$ or $<_i$ equal to 0 to find nearest to A and assign A to that Bound
$\text{Interval}_{=_i^u}$ match
Yes
No
Case 3
Yes
Interval_i Lower Bound = A
V_i is false; Interval_i Lower Bound = Interval_i Upper Bound = A; V_i = true
Interval_i Upper Bound = A
$\text{Interval}_{=_i^u}$ and $\text{Interval}_{=_k^l}$ match; Interval_k Lower Bound = Interval_i Lower Bound; V_i = false
End
Figure 4.2: Flow chart for insertions.

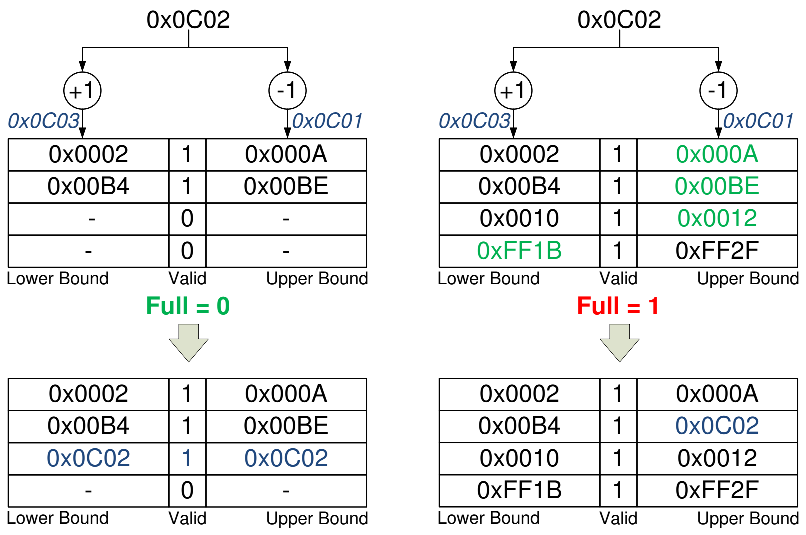


Figure 4.3: Example of insertion into the IF. Case 1.

upper bound so the lower bound of the interval k is set to the lower bound of i. If k is chosen to be invalidated then the upper bound of i is set to the upper bound of k.

Let's see some examples of insertions into the IF for the different cases described above. Figure 4.3 shows two examples for Case 1, where the incremented and decremented versions of the address to insert are not equal to any interval bound in the filter. On the left we can see the situation in which the filter has still some empty intervals, so that the *Full* signal is 0. In this case, the original address is inserted into both the lower bound and the upper bound of the first empty interval, whose valid bit is then set to 1. The figure on the right belongs to Case 1 as well. However, now the filter is full, and an existing interval must be widened. The bounds in green are those whose $>$ lines, for the lower bounds, or $<$ lines, for the upper ones, are set to 0. Such bounds are iterated to find the closest bound to the address to insert. This way, the search is shortened to half the number of bounds. In this case, the closest bound is 0x00BE, so the address is inserted in its place, thus yielding false positives for the addresses in between 0x00BE and 0x0C02.

Figure 4.4 depicts the last two cases. We can see, on the left, an insertion of Case 2. The incremented address matches a lower bound of a valid interval inserted in the filter, which means that the address to be inserted is contiguous

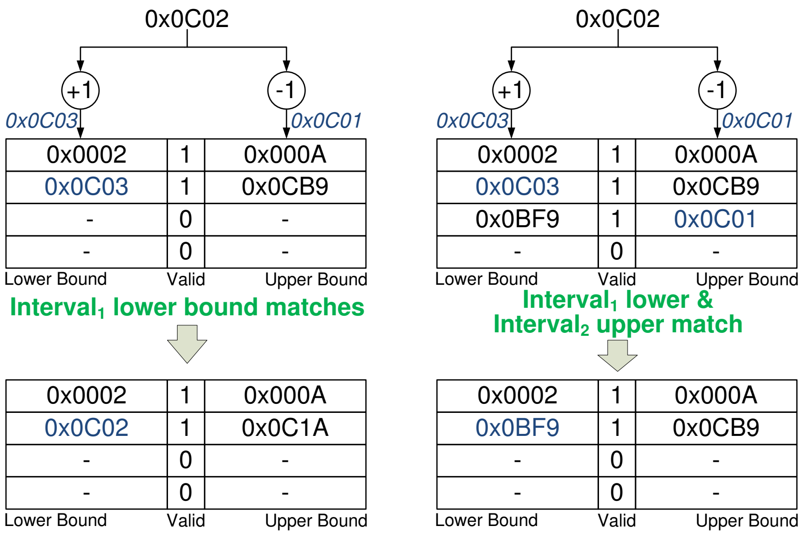


Figure 4.4: Example of insertion. Case 2 on the left. Case 3 on the right.

to an address that has already been inserted. Then, the original address (not incremented) replaces the lower bound, thus widening it without incurring false positives. On the right hand example of Figure 4.4, we can see an insertion of Case 3. The address to be inserted is the only one missing to merge intervals 1 and 2. The filter realizes because $=^l_1$ and $=^u_2$ signals are active. Then, interval 2 is invalidated by setting its valid bit to zero, and the lower bound of such interval is stored into the lower bound of interval 1.

4.2. Experimental Evaluation

This section is devoted to the experimental evaluation of the IF comparing its performance with that of a Bloom filter of similar hardware complexity.

The simulation environment used to evaluate the IF comprises the Simics full system execution-driven simulator and the GEMS's HTM Ruby module as described in Section 3.1.3. Ruby was modified to include the IF.

All workloads used for evaluating our proposal are part of the Stanford's STAMP suite discussed in Section 3.2. We have chosen the applications that lays emphasis on long-running transactions and large read and write sets. Such benchmarks are of special interest for signature evaluation since they put the

Table 4.1: Parameters and data set maximum and average sizes.

Bench	**Input**	**# xact**	*max* $\lvert RS\rvert$	*max* $\lvert WS\rvert$	*avg* $\lvert RS\rvert$	*avg* $\lvert WS\rvert$
Bayes	-v32 -r4096 -n2 -p20 -s0 -i2 -e2	536	2171	1631	81.8	45.1
Kmeans	-m40 -n40 -t0.05 -i random-n1024-d1024-c16	1380	134	65	99.7	48.5
Labyrinth	-i random-x48-y48-z3-n64	158	529	510	128.7	120.7
Yada	-a20 -i dots.2	1338	578	405	60.5	37.5

most pressure on signatures.

Synopsys and CACTI 5.3 [100] were used to estimate the area of the Interval Filter and the Bloom filter involved in the evaluation. A SRAM memory with 8-byte words and four separate read/write ports was modeled with CACTI to estimate the Bloom filter area. CACTI was also used to model a full-associative SRAM memory with 32-bit words and 2 banks for the IF. Additional control logic and extra comparators and incrementers used by the IF were modeled with Synopsis and were proven to have a small impact on the total area. Given an IF with $n = 10$ (i.e. ten intervals), the hardware-equivalent Bloom filter has 4 hash functions of the class H3 (H3 has proven better than others like Bit Selection [92]) and 2048 bits length. Both filters take about $0.09mm^2$ of die area each, using 65nm technology node. Hereinafter, results will be shown for an $n = 10$ interval filter compared to a 2048 bits, 4 hash function Bloom filter.

Experiments were carried out with 4 benchmarks from the STAMP suite: Bayes, Kmeans, Labyrinth and Yada. Such benchmarks exhibit the largest transaction data sets that cause Bloom filters to slowdown the execution because of false conflicts. Table 4.1 summarizes input parameters and the maximum and average RS/WS size in cache blocks for those benchmarks.

The motivation behind the Interval Filter comes from Figure 4.5. This figure shows the interval histograms for the four benchmarks. It is a classification by width of the intervals formed in the read sets and write sets of transactions, and it does not necessarily mean that every transaction in the system has intervals of every width, e.g. some transactions may have intervals of 2 and 10 instructions, but they might not have intervals of size 3, which would belong to a different transaction. All the benchmarks show some amount of single addresses, i.e. width-1 intervals, but most of addresses can be classified into intervals wider than 1 by extracting spatial locality features. In fact, the number of single addresses in the benchmarks is between 2% and 22% as shown in Table 4.2. To keep track of address sets as intervals instead of doing so as single addresses could save in

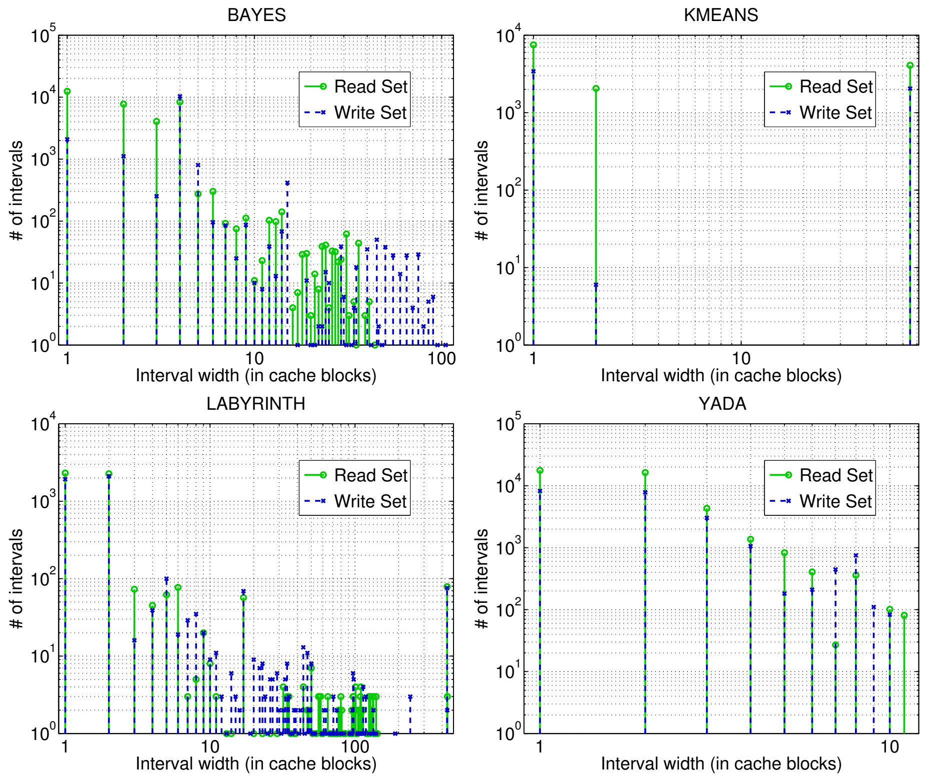


Figure 4.5: Number of intervals of different widths for Bayes, Kmeans, Labyrinth and Yada, both RS and WS (log scale).

Table 4.2: Percentage of single addresses.

Bench	**Number of single addresses**	
	Read Set	**Write Set**
Bayes	13.1%	2.7%
Kmeans	2.7%	2.5%
Labyrinth	4.2%	3.6%
Yada	22.0%	16.4%

space and performance.

Figure 4.7 shows the execution time of the Bloom filter versus the IF normalized to the perfect filter (i.e. infinite length, no false positives). Two cases can be observed:

- *Bayes and Yada:* The interval filter performs similar to or slightly worse than Bloom filters concerning these benchmarks. Two things cause such slowdown: (i) the high percentage of single addresses, see Bayes and Yada in Table 4.2, and (ii) transactions are made up of small-mid size intervals, as can be inferred from Figure 4.5, since the largest interval in Bayes is about 100 addresses in the figure, while the largest transaction is 2171 addresses (see Table 4.1). Also, for Yada, the largest interval is 11 addresses, while the largest transaction is 578 addresses. Therefore, having an interval filter with $n = 10$ intervals and a great amount of intervals to be stored in it, then "Case 1" (see Figure 4.1) will be the most frequent case of insertion, hence introducing lots of false positives.

  Another important fact to consider is the creation of the intervals. Figure 4.6 shows the interval creation in the write set of the largest transaction in each benchmark, i.e. it shows the result of having an infinite size interval filter in which addresses are inserted in order of appearance and the number of valid intervals are checked out after each insertion and then plotted. Notice that Bayes and Yada would need between 200 and 350 intervals to keep track of the whole set without false positives, however the interval filter size is 10 intervals. Flat parts in the Bayes curve corresponds to "Case 2" insertions.

- *Labyrinth and Kmeans*: The interval filter performs equally well or better than Bloom filters for these benchmarks. Now the number of single addresses is lower than Bayes and Yada (Table 4.2) and large transactions are made up of a few large-size intervals, since Figure 4.5 shows intervals greater than 400 addresses for Labyrinth while Table 4.1 shows maximum transactions about 500 and, 70 addresses intervals for Kmeans and maximum transactions of 70 and 130. Therefore, the interval filter does not get full immediately introducing few false positives. Figure 4.6 shows a flat creation of intervals for Kmeans while Labyrinth shows a rise and fall that corresponds to interval merging, "Case 3" insertions (see Figure 4.1).

  The behavior of these benchmarks is due to the data types they manage. Labyrinth makes a copy of a global multidimensional mesh inside a transaction which is represented as a multidimensional array. Kmeans keep a table of objects and attributes within an array. Conversely, Bayes and Yada use more complex and memory-scattered data structures as trees and lists.

The interval filter presents excellent results when the application has transactions with a given memory footprint. A few large intervals with a local access pattern get the best out of the IF, which may outperform the regular Bloom

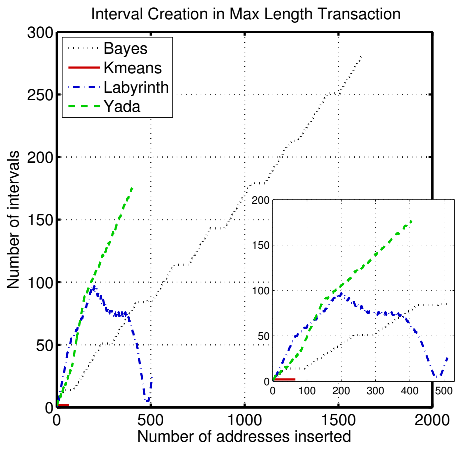


Figure 4.6: Write set interval creation for maximum length transactions of Bayes, Kmeans, Labyrinth and Yada.

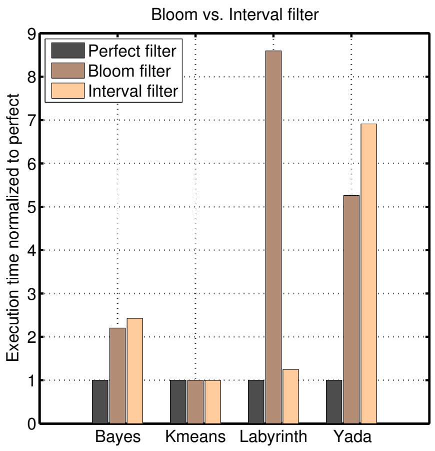


Figure 4.7: Execution time of Bayes, Kmeans, Labyrinth and Yada normalized to the perfect filter.

filter. Stride one accesses imply Case 2 insertions, which do not create new intervals and false positives either. Other strides may provoke Case 1 insertions causing false positives, as long as the filter is full. However, if the locations in between the stride are eventually accessed, such false positives might not harm the execution as the filter behaves as a conflict predictor. Also, Case 3 insertions enhance the performance of the IF as they reduce the population of intervals in the filter and do not yield false positives.

However, the IF harvests similar or worse performance than regular Bloom filters for data streams with poor locality features. Accessing many single addresses fills the filter early, so subsequent insertions falls into Case 1 category thus introducing false positives. Such a lack of generality in the behavior of the IF leads us to search other solutions like those described in next chapters. The IF could be thought of as a complement to Bloom filters to keep them from store large streams of consecutive addresses which could saturate the filter. It could be also useful in embedded systems where the access pattern is known in advance.

5 Locality-Sensitive Signatures

In this chapter we introduce *Locality-Sensitive Signatures* (LS-Sig) [79, 78, 80] as a Bloom-based signature optimization to conventional signatures.

Previous signature designs consider that all memory addresses are uniformly distributed across the address space. However, in real programs the address stream is not random as it exhibits some amount of locality. The main contribution of this chapter is a novel signature design based on Bloom filters, called LS-Sig, which exploits memory reference locality to reduce the probability of false conflicts. The proposal defines new maps for hash functions to reduce the number of bits inserted in the filter (occupancy) for those addresses with spatial locality. That is, nearby memory locations share some bits of the Bloom filter. As a result, false conflicts are significantly reduced in transactions that exhibit spatial locality in their read or write sets, but the false conflict rate remains unalterable for transactions that do not exhibit locality at all. This is favorable particularly for large transactions that usually present a significant amount of spatial locality. In addition, as the proposal is based on new locality-aware hash maps, its implementation does not require extra hardware.

We implement the proposed LS-Sig in a hardware TM simulator to show how savings in false conflicts translate into important performance improvements while executing concurrent transactions. In particular, different variants (hash maps) of LS-Sig were evaluated on codes from the STAMP [66] benchmark suite using the experimental environment described in Section 3.1.3. Also, these signatures were compared with other state-of-the-art implementations available in literature. In most cases, the results show significant improvements in performance, particularly for codes with large transactions.

The remainder of the chapter is organized as follows. In Section 5.1, the proposed LS-Sig design is introduced with a discussion of its basics, followed by its comparison with the generic signature designs for probabilistic evaluation. Section 5.2 deals with the implementation of different variants of LS-Sig on the Winconsin GEMS simulator, and discusses how LS-Sig can improve the execution performance in several cases.

5.1. Locality Sensitive Signatures

This section discusses how memory reference locality property can be used to reduce the probability of false conflicts in the signatures implemented as Bloom filters.

It is clarified that what will be considered in the ensuing discussion here is a Bloom filter (see Section 2.3.1) that maps a space of 2^n memory addresses, $N = \{0, 1, ..., 2^n - 1\}$, into an array of 2^m bits (indexes), $M = \{0, 1, ..., 2^m - 1\}$, $m \leq n$, through a family of k hash functions, $\{h_0, h_1, ..., h_{k-1}\}$. Hash functions of class H3 will be used, because they exhibit high quality behavior for memory address streams, as discussed in Section 2.3.2. Functions of class H3 basically define a linear transform between an n-bit word and an m-bit word: $h_i : GF(2)^{1 \times n} \rightarrow GF(2)^{1 \times m}$, $GF(2)$ being the Galois field of two elements [104], under the bitwise XOR. Two basic operations are defined over the Bloom filter: (i) inserting an address x by asserting its mapped bits ($h_i(x) = 1$), and (ii) checking if an address has been already inserted by testing if all its corresponding mapped bits are set to 1. Let $BF(x_0, x_1, ..., x_{q-1})$ be the set of asserted bits in the Bloom filter after inserting the sequence of q addresses $x_0, x_1, x_2, ..., x_{q-1}$. This set is given by $\bigcup_{i=0}^{q-1} BF(x_i)$, being $BF(x) = \bigcup_{j=0}^{k-1} h_j(x)$.

It is to be noted that false positives arise from two situations. First, an address y (not inserted) gives rise to a false positive if x was inserted in the Bloom filter and $BF(y) = BF(x)$, $y \neq x$. In such a case, one can say that x and y are *aliases*, that is, when we apply the hash functions, h_i, to these addresses, the resulting indexes are the same. In a Bloom filter, the probability of two addresses being aliases depends on particular hash functions and their number, k. For higher k, this probability would be smaller. Second, a false positive may appear because of current *occupancy* of the filter. This happens for a non-inserted address y, after the insertion of q addresses, if $BF(y) \subset BF(x_0, x_1, ..., x_{q-1})$ and y is not alias of any x_i. However, a false positive takes place. One expects that the higher the filter occupancy, the higher would be the probability of false positives. In fact, if the filter saturates (all bits set to 1) all subsequent queries for non-inserted addresses become false positives.

In general, small data set transactions, the common case [24], occupy a small fraction of the Bloom filter and, hence, show false positives most of which are due to aliases and only a few due to filter occupancy. However, large data set transactions show many false positives, which are not only due to aliasing but also to high filter occupancy. Reducing the number of hash functions, k, helps large data set transactions, but not the small ones [92]. Harnessing spatial locality could be a solution.

Memory reference locality is a property that may be used to favor small and large transactions simultaneously. In this chapter we propose to build a Bloom filter that maps locations far away from each other as normal Bloom filters do, while the nearby locations are mapped sharing some bits. This way, one can choose k to be high enough to favor small transactions, while large transactions will benefit from a reduction of the occupancy of the filter thanks to locality.

Different locality- or distance-sensitive hashing schemes are available in the literature. They are used to formulate queries of similarity in metric spaces using compact representations of objects [16, 47, 51] (see Section 2.3.5). Motivated by these definitions, a formal general signature scheme is introduced here that can take into account the locality of reference to reduce the occupancy of the filter when nearby addresses are inserted.

Definition 5.1.1 Let be a Bloom filter that maps a space of 2^n memory addresses, N, into a space of 2^m bits, M, $m \leq n$, through a family of k hash functions of the class H3, and let (N, d) and $(\wp(M), d_h)$ be two metric spaces. Such a Bloom filter is called (r, δ)-locality sensitive ((r, δ)-LS), with $r \in \mathbb{N}$ and $\delta : \mathbb{N} \rightarrow \mathbb{N}$, if, for any $x, y \in N$, it satisfies that,

- if $1 \leq d(x, y) \leq 2^r - 1$ then $0 \leq d_h(BF(x), BF(y)) \leq \delta(d(x, y)) < k$.

■

In a Bloom filter designed according to Definition 5.1.1, nearby locations assert not-disjoint bit sets into the bit array, i.e. they share some bits. The function d returns the distance between two addresses and may be considered as the value of the bitwise XOR, $d(x, y) = x \oplus y$, although the Euclidean distance, $d(x, y) = |x - y|$, can also be suitable. The usual metric of the distance between two sets, d_h, is the cardinality of the symmetric difference. Nevertheless, d_h is defined as $d_h(BF(x), BF(y)) = k - |BF(x) \cap BF(y)|$, which basically measures the number of differing hash function outputs when addresses x and y are mapped. As $|BF(x)| = k$, this metric is half of the cardinality of the symmetric difference between two sets.

Table 5.1: Example of locality-sensitive signature: addresses and their corresponding H3 indexes for a Bloom filter with k=4, 2^m=1024.

Address	h_0	h_1	h_2	h_3
0xffff0	240	158	889	554
0xffff1	586	158	889	554
0xffff2	90	347	889	554
0xffff3	736	347	889	554
0xffff4	181	906	484	554
0xffff5	527	906	484	554
0xffff6	31	591	484	554
0xffff7	677	591	484	554
0xffff8	718	497	62	163
0xffff9	116	497	62	163
0xffffa	612	52	62	163
0xffffb	222	52	62	163
0xffffc	651	741	675	163
0xffffd	49	741	675	163
0xffffe	545	800	675	163
0xfffff	155	800	675	163

It can be seen in Definition 5.1.1 that parameter r acts as the radius of action of the LS scheme. Addresses, whose distances are greater than $2^r - 1$, are mapped as though by a generic Bloom filter. The function $\delta(d(x, y))$, which returns the number of differing indexes between two addresses, $d_h(BF(x), BF(y))$, can as well be chosen to increase with $d(x, y)$, in such a way that the nearer the addresses are each other, the lesser disjoint are the sets of bits the addresses are mapped into.

Table 5.1 shows an example of an LS scheme where the outputs of the k hash functions were computed for a sequence of adjacent locations. Notice that for addresses with $d(x, y) = x \oplus y = 1$, their maps differ in only one value. Addresses with distance 2 are different in no more than 2 hash values. On the other hand, addresses with distances greater than $2^{k-1} - 1 = 7$ may have no hash values in common.

5.1.1. Implementation

This section introduces the implementation of a locality-sensitive signature scheme by defining the hash functions of its filters with specific parameters. Such an implementation is an instance of Definition 5.1.1, where $k = 4$, $d(x, y) = x \oplus y$,

$d_h(BF(x), BF(y)) = k - |BF(x) \cap BF(y)|$, $r = k - 1$ and:

$$\delta(d(x,y)) = \begin{cases} 1 & \text{if } d(x,y) = 1 \\ 2 & \text{if } 2 \leq d(x,y) \leq 3 \\ 3 & \text{if } 4 \leq d(x,y) \leq 7 \end{cases} \tag{5.1}$$

For operational reasons, function values outside the interval $[1, 2^r - 1]$ will be mapped as: $\delta(0) = 0$ and $\delta(d) = k$, if $d \geq 2^{k-1}$.

A Bloom filter with k hash functions is characterized by k H3 matrices $\{H_0, H_1, ..., H_{k-1}\}$, as seen in Section 2.3.2. Next, a set of H3 matrices are proposed according to the parameters described above.

In order to obtain an LS scheme that is compatible with the δ function in Expression 5.1, each hash function is transformed by masking its corresponding H_l matrix with the following square matrix, $M_l \in GF(2)^{n\times n}$, obtained by nullifying the last l elements of the identity matrix:

$$\begin{aligned} M_l &= diag(\overbrace{1, 1, \cdots, 1}^{n-l}, \overbrace{0, 0, \cdots, 0}^{l}) = \\ &= \begin{bmatrix} 1 & & & \cdots & \\ & 1 & & \cdots & \\ & & 1 & \cdots & \\ \underbrace{}_{n-l \text{ ones}} & & & \overbrace{}^{l \text{ zeros}} & \\ & \cdots & & 0 & \\ & \cdots & & & 0 \end{bmatrix}. \end{aligned} \tag{5.2}$$

The proposed locality-sensitive Bloom filter is thus characterized by k H3 matrices, $\{LH_0, LH_1, ..., LH_{k-1}\}$, where the last l rows of each matrix have been nullified, that is:

$$LH_l = M_l \cdot H_l. \tag{5.3}$$

Hereafter, the signature scheme that comprises filters with this type of hash functions will be referred to as LS-Sig.

Hence, the computation of an index by a hash function with matrix LH_l does not depend on the l least significant bits of the address. The example in Table 5.1 has been generated following this scheme. This way, the three last rows of the matrix associated with h_3, the two last rows of the matrix associated with h_2 and the last row of the matrix for the function h_1 are null, whereas no rows of the matrix associated with h_0 are null.

5.1.2. Features

Two important features of a hash function are uniformity and implementation cost.

A hash function is expected to be uniform, that is, inputs should be equitably mapped over the output range. This way, the number of inputs colliding on the same element of the destination space should be the same, avoiding aliases to concentrate on certain elements. In the case of a linear transformation, as the one in Expression 2.7, the alias distribution is directly related to the null space associated with the linear function. The null space is given by those vectors x such that $xH = 0$, 0 being the null vector of the destination space. It is commonly denoted as $N(H)$. If two inputs x, y are considered aliases, then $xH = yH \Rightarrow (x - y)H = 0$ and therefore, $x-y \in N(H)$. All linear combinations of vectors in the null space are aliases of 0, and an alias of a given input can be generated by adding a vector of the null space to it.

The rank of a matrix and the dimension of its null space are related by the rank-nullity theorem: $\dim(N(H)) + \text{rank}(H) = n$. According to this, for a given matrix H, it is desirable to have a null space as small as possible, because the number of aliases increases with the size of the null space. The minimum dimension of $N(H)$ is achieved for the maximum rank of the $n \times m$-matrix H, which will be m ($m \leq n$), if such a number of linearly independent rows in H is guaranteed. In this case $dim(N(H)) = n - m$.

Focusing on the computation of locality hash matrices according to Expression 5.3, it can be seen that the dimension of the null space, and consequently the uniformity of the hash function, will not be affected if the rows being nullified are linearly dependent on the remaining ones. This condition is easily fulfilled if the input space is larger than the destination space, as it has already been assumed ($m \leq n$), and the number of nullified rows does not exceed the aforementioned maximum kernel dimension, that is, $k \leq n - m$.

This situation was verified for all the matrices used in the experiments of this thesis. For example, the effect of nullifying the last rows on one of these matrices is shown in Figure 5.1, which depicts the hash matrix and its associated null space. Dots represent the non-zero matrix coefficients. For the null spaces, a vector basis is shown in the reduced row echelon form, which makes their visual comparison easy. It is to be noted that when nullifying the l-th last rows of the hash matrix, vectors with only the l-th bit asserted appear in the the null space basis. Nevertheless, it is important that the null space dimension remains at its maximum desirable value, $n - m$, for all LS matrices.

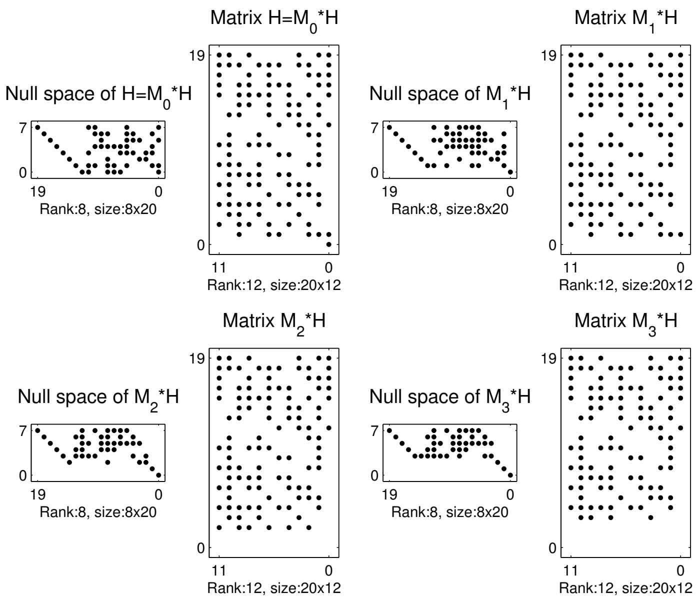


Figure 5.1: Effect of nullification of rows in the null space of an LS scheme's hash matrices ($N(H) \cdot H$ is the null matrix).

What is happening here is that the pattern of aliases is changing as rows are nullified, but the number of aliases remains unchanged. This fact can be observed in Figure 5.2, which shows the aliases of zero for the matrices in Figure 5.1. As the number of nullified rows increases, aliases concentrate in batches of consecutive addresses (nullifying l rows involves batches of size 2^l). This can be seen in the alias density histogram of the figure. Therefore, the way the locality sensitive hash matrices are generated does not alter the null space dimension, and consequently the number of aliases will be the same as that of the original hash functions.

As regards implementation costs, no additional hardware is required, because the proposed LS-Sig can be considered a special case of Bloom filters. Besides, the XOR trees could be even simpler, because several hash functions do not make use of certain bits of the address. LS-Sig can also be implemented directly following a parallel Bloom organization [92]. In addition, this locality-sensitive scheme can be easily combined with or extended to other implementations, like the PBX hashing [111], as shown in Section 5.2.4.

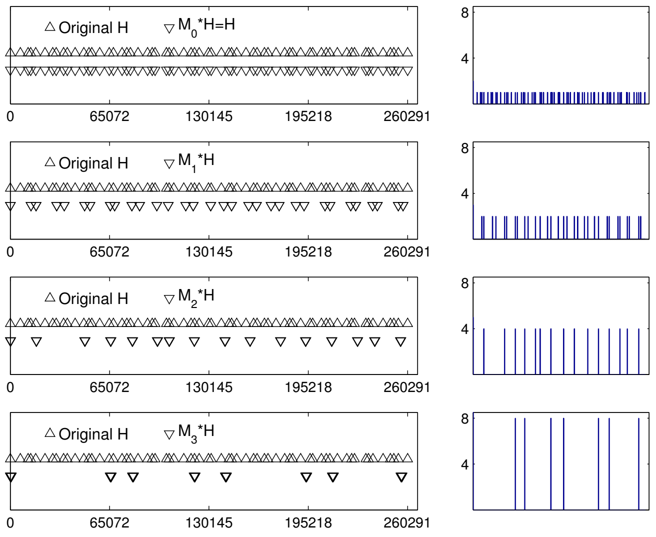


Figure 5.2: Pattern of aliases of zero for the LS matrices in Figure 5.1 (left), and its density distribution (right).

5.1.3. False Positive Analysis

Next, a general expression of the false positive rate, similar to Expression 2.3 in Section 2.3.1, is obtained for the locality-sensitive signatures of Definition 5.1.1 with the δ function of Expression 5.1 and the rest of conditions in Section 5.1.1.

The probability of false positives is expected to depend on the spatial proximity between addresses. For instance, a new address y is considered for insertion in the Bloom filter, x being the nearest address already inserted. Considering the stochastic variable t, assume that f_t is the probability that δ function is equal to t, for any address to be inserted with its nearest address already in the filter,

$$f_t = Pr(\delta(d(y, \min_{x \text{ inserted}} d(x,y))) = t), 1 \le t \le k. \tag{5.4}$$

Here, $\min_x f(x)$ denotes the value of x where a given function $f(x)$ is minimum. This definition excludes the repetitions of addresses during insertion; hence, f_0 has not been taken into account and consequently t goes from 1 to k, as δ definition saturates beyond k.

This way, the probability of filter bits being zero for distant inserted addresses

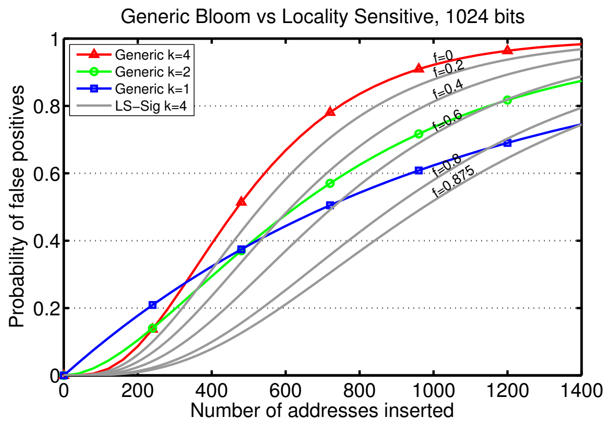


Figure 5.3: Probability of false positives of generic and LS signatures varying the parameter $f = \sum_{t=1}^{3} f_t$ (the higher the f, the more the locality).

continues to follow the Expression 2.1. Nevertheless, neither of the two near addresses x, y (having distance less than 2^{k-1}) will assert k bits. Instead, they assert only $k + \delta(d(x, y))$ bits in total. Making use of the probabilities of Expression 5.4, the probability of a bit remaining zero for the locality-sensitive scheme can be written thus:

$$p_{\text{ZERO}}^{\text{local}}(m, q, k) = \left(1 - \frac{1}{2^m}\right)^{q \sum_{t=1}^{k} t \cdot f_t}. \tag{5.5}$$

Hence, the probability of false positives for the locality-sensitive signature can be expressed as follows:

$$p_{\text{FALSE POSITIVE}}^{\text{local}}(m, q, k) = \left(1 - \left(1 - \frac{1}{2^m}\right)^{q \sum_{t=1}^{k} t \cdot f_t}\right)^{k}. \tag{5.6}$$

Figure 5.3 shows the analytical evaluation of false positive probability for the original Bloom filter (Expression 2.3) with several k values, and the proposed LS scheme (Expression 5.6) for $k = 4$. To parameterize the evaluation, $f = \sum_{t=1}^{k-1} f_t$ was introduced as the probability of an address being near to some inserted address. Consequently, $1 - f = f_k$ is the probability of being far from those already in the filter. As the parameter f gathers probabilities for different distances, a Zipf distribution was chosen, which is often assumed for modeling locality of reference [27, 52]. In this way, Expression 5.6 was evaluated with

Table 5.2: Probabilities defined by Expression 5.4 for STAMP codes (the higher the f_1 the more the spatial locality in transactions).

Benchmark	f_1^{RS}	f_2^{RS}	f_3^{RS}	f_4^{RS}	f_1^{WS}	f_2^{WS}	f_3^{WS}	f_4^{WS}
Bayes	0.32	0.25	0.14	0.29	0.40	0.29	0.14	0.17
Genome	0.24	0.15	0.14	0.47	0.26	0.20	0.03	0.51
Intruder	0.24	0.15	0.07	0.54	0.17	0.14	0.09	0.60
Kmeans	0.48	0.25	0.12	0.15	0.49	0.25	0.12	0.14
Labyrinth	0.46	0.25	0.13	0.16	0.46	0.26	0.13	0.15
SSCA2	0.19	0.06	0.13	0.62	0.14	0.00	0.00	0.86
Vacation	0.13	0.10	0.08	0.69	0.29	0.12	0.19	0.40
Yada	0.25	0.24	0.16	0.35	0.29	0.27	0.17	0.27

$f_2 = \frac{1}{2}f_1$, $f_3 = \frac{1}{3}f_1$. It is to be noted that Expressions 5.5 and 5.6 are valid only for the distance metric under consideration. Likewise, this metric introduces some constraints to possible f_t values. Considering a monotonically increasing consecutive sequence of different addresses, it was thus fulfilled that $f_4 \geq \frac{1}{8}$, and $f_1 \leq \frac{1}{2}$. The lower bound of f in the plot was derived from these constraints.

Whereas low values of k are advantageous for large transactions, and high values of k for small ones, with a generic Bloom filter, it can be inferred from Figure 5.3 that the LS scheme can achieve the benefits of both situations if the address sequence exhibits medium/high spatial locality.

5.1.4. Locality in Benchmarks

Prior to including locality-sensitive signatures in a cycle accurate HTM simulator, a straightforward functional TM system was developed to estimate the locality properties of the benchmarks used in evaluating the present proposal. Intel's PIN instrumentation tool [57] was used to rapidly implement the system, which we have called TMtool, and it is described in Section 3.1.4.

The probabilities f_t, $1 \leq t \leq k = 4$, were measured for both the read and the write sets of different benchmarks (see Table 5.2). As expected, exploitable locality was present in all codes. In fact, for two of the codes, Kmeans and Labyrinth, f_1 almost reached its maximum $\frac{1}{2}$, for the metric under consideration.

Using our PIN TMtool simulator, the filter occupancy (i.e. the number of bits set to 1 in the filter) of committed transactions was recorded for both generic and LS signatures. The average occupancy saving percentage of LS-Sig over generic signatures was calculated as $saving = \frac{\text{occupancy\_generic} - \text{occupancy\_locality}}{\text{occupancy\_generic}}$ and shown in Figure 5.4. The results were obtained by varying the filter size

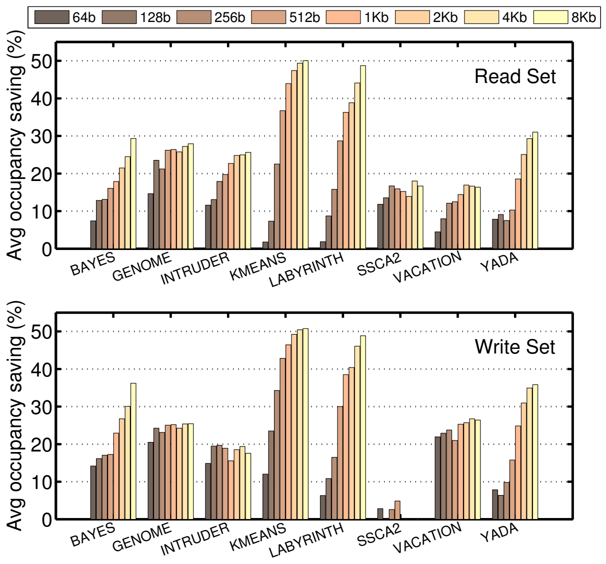


Figure 5.4: Average Bloom filter occupancy saving of locality-sensitive signature with respect to generic signature, both for read set (top) and write set (bottom), varying the filter size from 64b to 8Kb.

from 64Kbit to 8Kbit. On large filters, occupancy conflicts are rather unlikely; therefore, rightmost bars show the maximum possible occupancy savings that LS-Sig can achieve for the tested benchmarks. It is to be noted that codes with the highest locality in Table 5.2 get the highest savings. On the other hand, SSCA2 benchmark hardly saves bits in the signatures because its transactions are very small. In fact, its WS is of 1 or 2 addresses on average (see Table 5.4) and the saving is 0% as shown in Figure 5.4. Finally, occupancy saving helps on diminishing the probability of stalling or aborting transactions due to false positives. Hence, these significant occupancy savings of LS-Sig are expected to improve the execution time.

5.2. Experimental Evaluation

This section deals with the simulation methodology (Section 5.2.1), and the experimental results obtained from the simulator (Sections 5.2.2 to 5.2.5). Section 5.2.2 explores different LS-Sig schemes, Section 5.2.3 discusses about false

Table 5.3: Input parameters for the benchmarks

Benchmark	**Input**
Bayes	-v32 -r1024 -n2 -p20 -s0 -i2 -e2
Genome	-g512 -s64 -n8192
Intruder	-a10 -l128 -n128 -s1
Kmeans	-m40 -n40 -t0.05 -i rand-n1024-d1024-c16
Labyrinth	-i rand-x32-y32-z3-n64
SSCA2	-s13 -i1.0 -u1.0 -l3 -p3
Vacation	-n4 -q60 -u90 -r16384 -t4096
Yada	-a20 -i 633.2

sharing and LS-Sig, Section 5.2.4 describes the effect of applying PBX hashing on LS-Sig, and finally, Section 5.2.5 discusses about saving hardware with our proposal.

5.2.1. Methodology

We followed the methodology outlined in Chapter 3 to evaluate our LS-Sig designs. Ruby was modified to include the locality-sensitive signature designs. For implementing the hash functions, same H3 matrices of Ruby were used after effecting the modifications described in Section 5.1.1.

Simulation experiments use perfect signatures (no false positives, hardware unimplementable) as the goal to reach. Parallel signatures, both generic and locality-sensitive ones, ranging in length from 64 bits to 8K bits[1], were used to gain a comprehensive insight into locality-sensitive signatures behavior. All signatures used 4 hash functions.

The proposed signature schemes were experimentally evaluated using all the codes of the STAMP suite (see Section 3.2). Table 5.3 summarizes the input parameters for the benchmarks, and Table 5.4 shows the main transactional characteristics of the workloads. Column "#xact" is the number of committed transactions, and column "Time in xact" shows the percentage of transactional cycles. The metric for locality in benchmarks, column "Xact locality", was drawn from Table 5.2. The last four columns stand for average and maximum values of RS and WS size distributions, measured in cache blocks.

[1]64 bits matches the word length in SPARC architecture whereas 8K bits matches the performance of perfect signatures for the simulated benchmarks

Table 5.4: Workload transactional characteristics. (Data set sizes in the right-most four columns are reported in cache blocks)

Bench	**#xact**	**Time in xact**	**Xact locality**	***avg* $\|RS\|$**	***avg* $\|WS\|$**	***max* $\|RS\|$**	***max* $\|WS\|$**
Bayes	523	94%	High	76.9	40.9	2067	1613
Genome	30304	86%	Mid	12.1	4.2	400	156
Intruder	12123	96%	Mid	19.1	2.5	267	20
Kmeans	1380	6%	High	99.7	48.5	134	65
Labyrinth	158	100%	High	76.5	62.9	278	257
SSCA2	47295	19%	Low	2.9	1.9	3	2
Vacation	24722	97%	Mid	19.7	3.6	90	30
Yada	5384	100%	High	62.7	38.4	776	510

5.2.2. Exploring Radius and Delta

Certain works [79, 92] consider that parallel signatures should be used, instead of regular ones, because they can perform equally well or even better, besides being more area-efficient. Another suggestion they offer is using four or more high-quality hash functions, preferably from the H3 family. Actually, Sanchez [91] proves that, in most cases, the performance given by four hash functions is better than that given by one or two functions. Besides, using eight hash functions gives no significant improvement and it requires additional hardware. Consequently, all experiments were carried out using parallel Bloom filters with 4 H3 hash functions.

Complying with Definition 5.1.1, six (r, δ)-LS signatures are explored next by combining two different delta functions, δ_0 and δ_1, with three different radii, 1, 3 and 5. Also, another delta function, δ_P, is proposed, which behaves better independently of benchmarks. Its radii are 3 and 5. So, the following are the explored LS schemes:

1. (r, δ_0)-LS: Addresses within intervals of radius r (i.e $[0, 2^r - 1], [2^r, 2^{r+1} - 1], ...$) are mapped to the same bits in the filter. That is, 0 indexes are different between the maps of the addresses within the same interval:

$$\delta_0(d(x, y)) = 0 \quad \text{if } 1 \leq d(x, y) \leq 2^r - 1.$$

2. (r, δ_1)-LS: Addresses within intervals of radius r are mapped to the same bits except one. That is, when mapping the addresses of an interval, only 1 index differs between maps:

$$\delta_1(d(x, y)) = 1 \quad \text{if } 1 \leq d(x, y) \leq 2^r - 1.$$

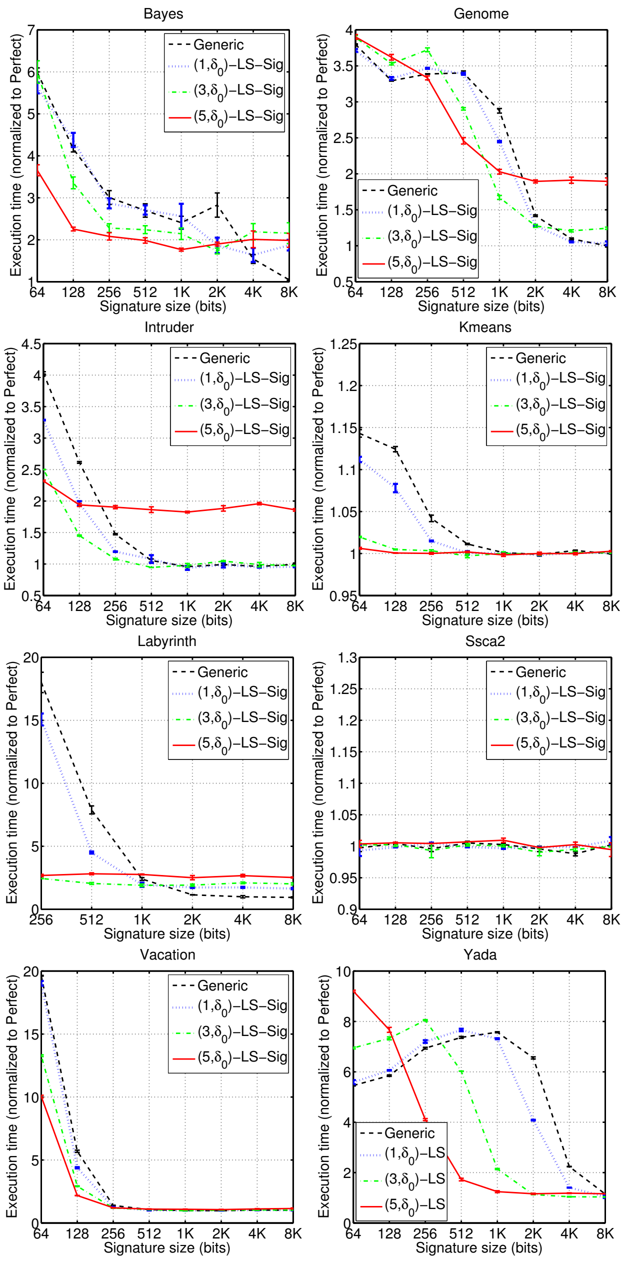


Figure 5.5: Execution time normalized to perfect signature comparing parallel generic signatures and (r, δ)-LS-Sig with $r \in \{1, 3, 5\}$ and δ_0.

3. (r, δ_P)-LS: Addresses within intervals of radius r are mapped depending on the distance between them. Thus, delta function is defined piecewise as follows:
$$\delta_P(d(x,y)) = \begin{cases} 1 & \text{if } d(x,y) = 1 \\ 2 & \text{if } 2 \le d(x,y) \le 2^{\lceil \frac{r}{2} \rceil} - 1 \\ 3 & \text{if } 2^{\lceil \frac{r}{2} \rceil} \le d(x,y) \le 2^r - 1 \end{cases}$$
For example, for $r = 5$, δ_P is as follows:
$$\delta_P(d(x,y)) = \begin{cases} 1 & \text{if } d(x,y) = 1 \\ 2 & \text{if } 2 \le d(x,y) \le 7 \\ 3 & \text{if } 8 \le d(x,y) \le 31 \end{cases}$$

Figure 5.5 shows the execution time normalized to perfect filters for all the benchmarks using locality-sensitive signatures with δ_0 and three different radii. Time for parallel generic signatures (Generic) is also shown for comparison. (r, δ_0)-LS-Sig operates as a generic filter which maps the addresses at granularity higher than that of cache blocks. Thus, δ_0 could suffer from additional false sharing (see Section 5.2.3) when the radius is high.

For radius equal to 1, the memory block is 1 bit larger from the viewpoint of signatures. Then, $(1, \delta_0)$-LS-Sig introduces little additional false sharing and consequently almost all benchmarks behave similar to parallel generic signatures when the filters are large, except for Labyrinth and Bayes, which performs worse, because of false sharing, with about 1.7× slowdown for 2Kb, 4Kb and 8Kb in Labyrinth and 1.7× slowdown for 8Kb in Bayes. With shorter filters, $(1, \delta_0)$-LS-Sig performs equally well or slightly better due to occupancy saving. For radius equal to 3, the signature block is larger by 3 bits; so, there is more additional false sharing, as shown by larger filters. It can be seen from Figure 5.5 that Labyrinth slows down by 2× for 4Kb and 8Kb signatures, Bayes by 2× for 8Kb signature, and Genome by 1.25× for 8Kb signature. Conversely, occupancy savings lead to better results when the signature size decreases: Bayes shows a speedup of 1.25× over the generic scheme from 128b to 2Kb signatures, Genome 1.7× for 1Kb, and Intruder and Vacation 1.5× on average for small signatures, while Labyrinth and Yada speed up by 7× and 5× respectively. When the radius is equal to 5, 2^5 addresses are mapped to the same four bits in the signature. Consequently, false sharing results in 2× slowdown for large signatures in Bayes, Genome, Labyrinth and Intruder. However, the sparse filling of the filter leads to significant results in almost every benchmark for small signatures. In most cases, Labyrinth, Yada, Bayes, Intruder and Vacation are twice as fast as parallel generic signatures, as shown in Figure 5.5, whereas Yada and Labyrinth are six-times as fast in some cases.

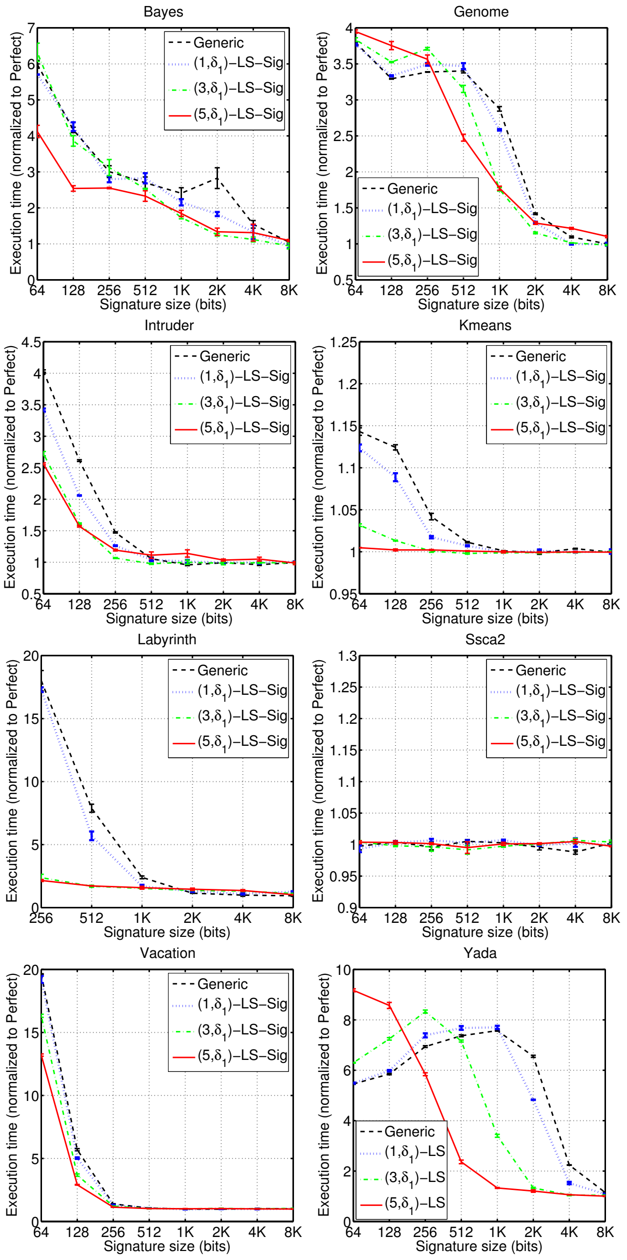


Figure 5.6: Execution time normalized to perfect signature comparing parallel generic signatures and (r, δ)-LS-Sig with $r \in \{1, 3, 5\}$ and δ_1.

Figure 5.6 shows the execution time normalized to perfect filters for all the benchmarks using (r, δ_1)-LS-Sig with $r \in \{1, 3, 5\}$. δ_1 combines three hash functions operating at granularity coarser than that of cache and one hash function working at cache block granularity. This way, the hash function working at cache granularity can distinguish between addresses mapped to the same bits because of coarser granularity of the other three hash functions, as long as it does not incur a false positive. As Figure 5.6 shows, benchmark performance for large signature sizes is now either slightly worse than parallel signatures or the same. With $r = 5$, Genome slows down its execution by 1.1× for 4Kb and 8Kb signatures, Intruder by 1.2× for 1Kb and Labyrinth 1.36× for 4Kb. For small signatures, execution is slower than that of δ_0, because of the addition of false positives from the hash function that operates at cache granularity in addition to the false sharing from the three hash functions operating at coarser granularity. Even so, δ_1 outperforms generic signatures in most cases.

Additionally, we have defined one more LS scheme that performs more evenly, whatever be the benchmark: (r, δ_P)-LS. Such a scheme merges several granularities to trade off between additional false sharing and false positives. Figure 5.7 shows the execution time normalized to perfect signatures for parallel generic signatures, (r, δ_P)-LS-Sig with $r \in \{3, 5\}$ and $(5, \delta_1)$-LS-Sig. It is to be noted that (r, δ_P)-LS-Sig performs similar to or better than generic filters on large signature sizes, since they can get rid of the additional false sharing effect that comes up in δ_1 and δ_0 for Bayes, Genome, Intruder or Labyrinth. As far as small signatures are concerned, $(3, \delta_P)$-LS-Sig gets closer to generic filters because it is built up with smaller radius than that of $(5, \delta_P)$-LS-Sig, which, conversely, gets closer to $(5, \delta_1)$-LS-Sig.

From the results of (r, δ_P)-LS-Sig, shown in Figure 5.7, benchmarks can be classified into three groups, considering their behavior:

1. *SSCA2:* This benchmark exhibits the smallest transactions of the whole suite. RS and WS maximum sizes are only 3 and 2 cache blocks respectively. Moreover, the benchmark spends most of the time outside transactions (see Table 5.4). Hence, SSCA2 is not signature-dependent.

2. *Kmeans, Vacation, Intruder and Labyrinth:* These benchmarks show similar behavior when signature size decreases. In some cases, $(3, \delta_P)$-LS-Sig and $(5, \delta_P)$-LS-Sig reduce the execution time considerably. They always either outperform or match the performance of generic signatures.

   Kmeans is low contended and spends only 6% of time in transactions; so, this is also not signature-dependent (Figure 5.7 shows a speedup of 1.14x for the best case). Even so, (r, δ_P)-LS-Sig reduces execution time of generic ones when 128b filters are used, because transactions are of medium size and exhibit high locality (see Table 5.2).

In Vacation, (r, δ_P)-LS-Sig matches the execution time of generic signatures, because of mid-locality and medium-to-small scale transactions. They are better for 64 and 128b signatures, because the maximum data set size (max RS size is 90) is close to the signature size, and generic signatures have higher occupancy. Vacation is high contended and does not scale well for small signatures. Intruder behaves the same way as does Vacation, but scales better.

Finally, Labyrinth shows a great improvement in performance. High locality, large transactions on average (both RS and WS) and high contention, help (r, δ_P)-LS-Sig in outperforming generic signatures remarkably.

3. *Bayes, Genome, Yada:* In these benchmarks, LS-Sig yields better results than generic signatures for large signature sizes, but slightly worse results for small ones. This behavior is related to the manner in which LogTM-SE resolves conflicts. LogTM-SE stalls transactions that request for a conflicting address, retries its coherence operation, and aborts on a possible deadlock cycle. Hence, with (r, δ_P)-LS-Sig, transactions can run for a longer time before encountering a conflict, even on small signatures, but on abort, they must undo the log that is longer than it would be if the conflict had been detected earlier. In some cases, a false conflict can behave as a conflict prediction and it can accelerate progress [19]. Figure 5.8 shows the average RS and WS percentages of false positives for Genome and Yada. The percentage of false positives was obtained by dividing the total number of false positives by the total number of both false and true positives. Figure 5.7 shows that the number of false positives is higher for signatures from 64b to 256b, but Figure 5.8 shows that the number of true positives also is higher as the percentage becomes low. This confirms that transactions run longer by virtue of (r, δ_P)-LS-Sig. It is to be noted that decreasing false positive rate in signatures does not necessarily lead to direct improvement in performance, as other factors, like abort patterns, also matter.

   In real systems, signatures are likely to increase in size, as in the case of caches and memories. This way, despite the aforementioned execution time loss when signatures are 256b or smaller, $(3, \delta_P)$-LS-Sig and $(5, \delta_P)$-LS-Sig are good alternatives to parallel generic signatures.

Finally, the linear correlation between execution time and abort time was calculated for the experimental setups of Figures 5.5, 5.6 and 5.7. The total time

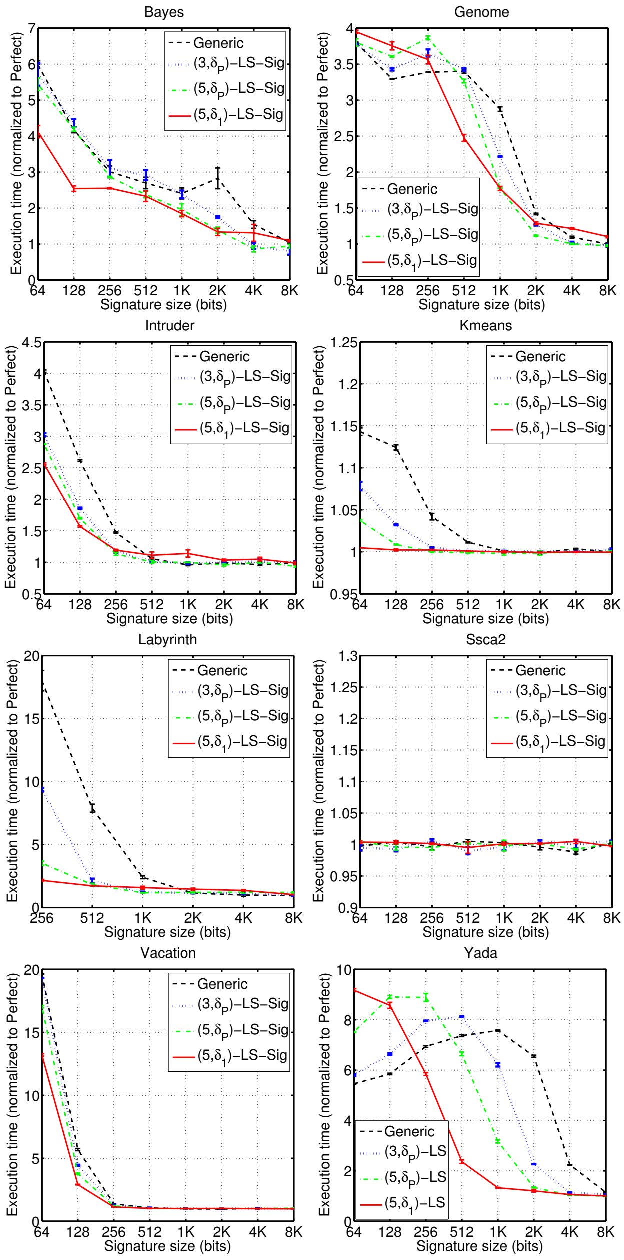


Figure 5.7: Execution time normalized to perfect signature comparing parallel generic signatures and (r, δ)-LS-Sig with $r \in \{3, 5\}$ for δ_P and $r = 5$ for δ_1.

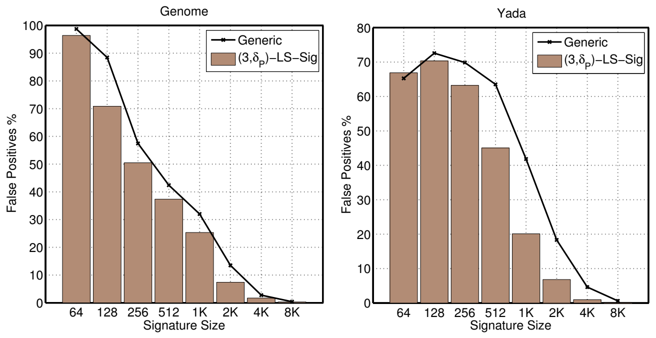


Figure 5.8: Average of RS and WS percentage of false positives for generic and (3,δ_P)-LS signatures.

of aborts depends on the number of aborts, as also of the time required to solve them, which is mainly a function of the length of the log to be undone. The observed correlation coefficient varies from 0.88 to 0.99, which reflects the strong correlation between the execution time and the abort time.

Implication: For large signatures, (r, δ_P)-LS-Sig performs equally well or better than parallel generic signatures, while for small signatures, in most cases, it outperforms parallel ones. Also, δ_P behaves more evenly than δ_0 and δ_1 for all the benchmarks tested. Therefore parallel (r, δ_P)-LS-Sig is a good alternative to parallel generic signatures.

5.2.3. Interthread vs. Intrathread Locality: The Effect of Additional False Sharing

In the preceding section, the term *additional false sharing* has been introduced to denote the false sharing added by locality-sensitive signatures because of hash coarse granularity. This section discusses how additional false sharing can affect the performance of benchmarks and its relationship to locality.

To break down the number of false positives into those caused by Bloom aliasing/occupancy and those caused by additional false sharing, the following procedure was followed depending on the delta function (*super block* will be used to refer to locality-sensitive hashing blocks):

- δ_0, *every hash function at the same granularity, larger than cache blocks*: An additional false sharing false positive is detected if the address involved

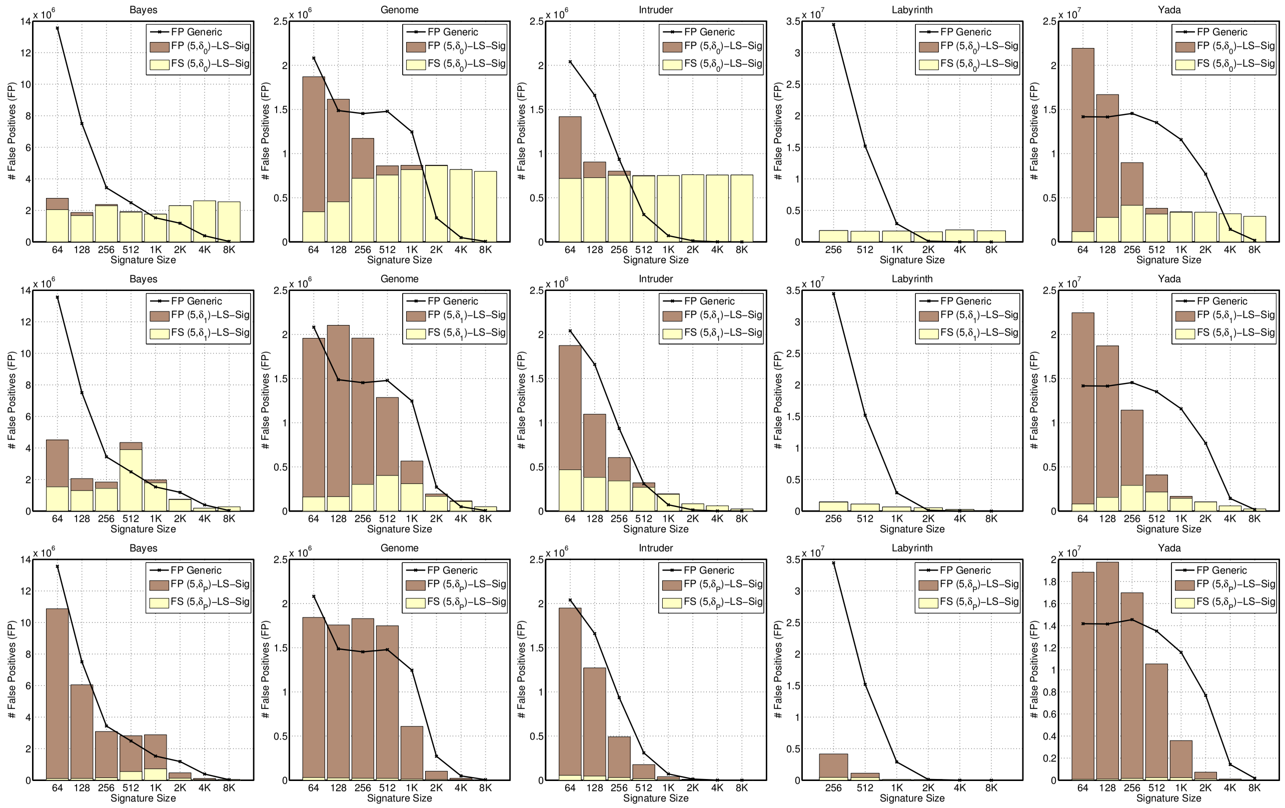


Figure 5.9: Number of false positives of parallel generic signatures (FP Generic) compared to the number of false positives of $(5, \delta_0)$-LS-Sig (top row), $(5, \delta_1)$-LS-Sig (middle row) and $(5, \delta_P)$-LS-Sig (bottom row). False positives for LS-Sig are broken down into false positives due to aliasing (FP) and false positives due to additional false sharing (FS).

was not really inserted in the filter, but another address of the same signature super block was inserted earlier. The filter thus matches the address because of additional false sharing.

- δ_1*, one hash function at cache block granularity and the others at the same larger granularity*: A false positive due to additional false sharing is detected if there is a false positive due to aliasing/occupancy in the subfilter which operates at cache block granularity, and the address was not really inserted, but another one in its super block was.
- δ_P*, one hash function at cache granularity and the others at larger and different granularities*: A false positive due to additional false sharing is detected if there is a false positive due to aliasing/occupancy in the filter which operates at cache block granularity, and all the other filters operating at different and coarser granularity result in a false sharing false positive.

False sharing due to cache blocks was not taken into account, because benchmarks were tuned to avoid it via padding (see Section 3.2.2).

Figure 5.9 shows the number of total false positives (FP), i.e. the number of false positives in the read set of all transactions plus the number of false positives in the write set of all transactions, for parallel generic signatures, and the number of false positives for $(5, \delta_0)$-LS-Sig, $(5, \delta_1)$-LS-Sig and $(5, \delta_P)$-LS-Sig. False positives concerning LS-Sig are broken down into false positives due to signature aliasing/occupancy (FP) and those due to signature false sharing (FS). The figure shows five benchmarks that are sensitive to signature false sharing: Bayes, Genome, Intruder, Labyrinth and Yada.

It needs to be noted that δ_0, the top row, is the locality-sensitive hash function that shows more false positives due to additional false sharing. In this case, when the filter size is large, the number of false positives due to occupancy is close to zero and almost every false positive is due to false sharing (light yellow in Figure 5.9). This situation leads to the definition of two types of locality in a parallel benchmark and therefore, to the classification of benchmarks depending on the type of locality they exhibit:

- *Interthread locality*: A parallel code shows interthread locality, if their threads reference shared memory locations near in time and there is spatial proximity between such locations.

  Figure 5.5 shows that Bayes and Labyrinth slow down their execution for $(r \in \{1, 3, 5\}, \delta_0)$-LS 8Kb signatures, implying thereby that such benchmarks exhibit high interthread locality. Genome and Intruder get worse results with radius 5, but with radius 1 or 3 they do not slow down their execution significantly. Thus, shared data is located farther away from each other than that of Bayes and Labyrinth; so, it can be said that Genome and Intruder exhibit medium interthread locality. The other benchmarks, which do not show significant slowdown for $(5, \delta_0)$-LS-Sig, can be regarded as of low interthread locality. Overall, interthread locality is likely to be a feature of codes whose shared data comprise arrays of primitive data types and light-weight structures, like the Labyrinth's three dimensional integer array which represents the maze where the benchmark finds the shortest-distance paths between pairs of points, or the net of structs which holds the marks of the Bayesian network in Bayes. However, the shared data access patterns will ultimately determine the degree of interthread locality in a parallel application.

- *Intrathread locality*: A parallel code shows intrathread locality if their threads reference their private memory locations near in time and there is spatial proximity between locations.

Therefore, codes not showing much interthread locality could benefit from (r, δ_0)-LS-Sig with high radius, because lots of adjacent locations will be mapped to the same bits in the filter, keeping occupancy low. However, codes with high interthread locality may need lower radius and other delta locality functions as discussed below.

The middle row of Figure 5.9 shows the total number of additional false sharing false positives for $(5, \delta_1)$-LS-Sig, which has a filter that operates at cache granularity. It avoids false positives due to false sharing in other filters as long as a false positive due to occupancy is not detected in that filter. If it is detected, the other filters may get a false positive due to false sharing. The probability of getting false positives increases with the occupancy of the filter; so, the probability of getting false positives due to false sharing will be higher when the subfilter operating at cache granularity gets full. However, thanks to locality-sensitive signatures, three more subfilters, operating at higher granularity, have not been spoiled by occupancy. Furthermore, by the time the subfilter operating at cache granularity has been spoiled by occupancy, there will be lots of addresses in the signature that may probably form blocks of contiguous, locality-wise addresses, so that the other filters that have been storing super block addresses do not get lots of false positives due to false sharing, because almost every address in a super block will probably have been inserted. This is better achieved by functions of class δ_P, because they map addresses at different granularities, both fine and coarse. Such a scheme minimizes the number of false conflicts due to false sharing as shown by the bottom row graphs in Figure 5.9.

Implication: Benchmarks showing low interthread locality benefit from LS-Sig with large radii, and vice-versa. As regards intrathread locality, the larger the radius the better it would be, regardless of the amount of locality, because only private data is involved. In case of general purpose systems, where workloads may vary widely and the degree of interthread locality is not known in advance, (r, δ_P)-LS-Sig is the best choice as it involves a trade-off between large radii, small radii, δ_0 and δ_1 LS-Sig.

5.2.4. Locality-Sensitive PBX Hashing

In this section, Page-Block-XOR (PBX) hashing [111] is used to reduce hardware costs of Locality-sensitive Signatures. PBX hashing was devised by Yen et al. to keep the randomness of H3 functions while saving in area, power and latency of the signature implementation. H3 hashing requires a tree of XOR gates per bit of the hashing function output index, conversely, PBX requires only one XOR gate per bit resulting in a single XOR gate row per hash function.

The insight behind PBX is that the input bits have enough randomness to minimize the XOR trees involved in the index computation. Such randomness is calculated using the entropy of the addresses. Assuming 32-bit virtual and physical addresses, cache-blocks of 64 bytes and page size of 4kB, the highest entropy (i.e. higher randomness) was found from the 26th to the 6th bit of the addresses, for the STAMP benchmarks. Yen et al. also found that bit-field overlap leads to higher false positive due to correlation between bit-fields. Therefore, the physical page number bits, from bit 26th to bit 17th, were combined with the cache-index bits[2], from bit 16th to bit 6th. Figure 5.11 shows the PBX binary matrices used in this chapter. Such matrices define a surjection, that is, for every value into the codomain of indexes there exists at least one value in the domain of addresses. To assure that, the rank in $GF(2)$ (the Galois field of two elements) of the matrices was calculated and proved to have full rank. To get the LS-Sig PBX matrices, they were shifted downwards inserting blank rows on the top, thus maintaining the bit-field disjunction and ensuring maximum use of bits. However, as signature gets smaller, the leftmost columns were subtracted from the matrices, keeping the bits in those columns from being used to compute the final index. This way, LS-Sig PBX could yield slightly different results for some benchmarks (e.g. Yada, Labyrinth) when signatures are small as shown in Figure 5.10.

[2] The size of the bit-fields are not tied to the configuration of the CMP [111]

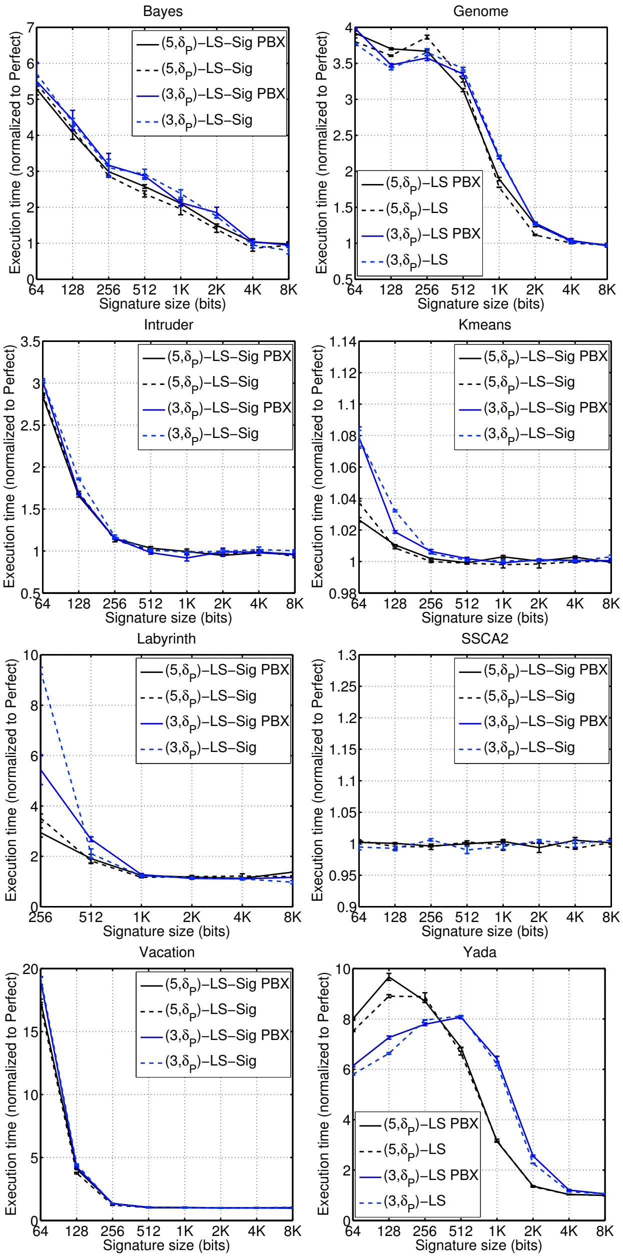


Figure 5.10: Execution time normalized to perfect signature (no false positives) comparing Generic and $(r \in \{3, 5\}, \delta_P)$-LS-Sig to their PBX versions.

$$
\begin{array}{cccc}
H_0^{PBX} & H_1^{PBX} & H_2^{PBX} & H_3^{PBX} \\
\left(\begin{array}{c}
00000000001 \\ 00001000000 \\ 00000000010 \\ 00000100000 \\ 00000000100 \\ 00000010000 \\ 00000001000 \\ 01000000000 \\ 00100000000 \\ 10000000000 \\ 00010000000 \\ \hline
00000000001 \\ 00010000000 \\ 00000000010 \\ 00001000000 \\ 00000000100 \\ 00000100000 \\ 00000001000 \\ 00000010000 \\ 00100000000 \\ 01000000000
\end{array}\right) &
\left(\begin{array}{c}
00000001000 \\ 00000000001 \\ 00000000100 \\ 00000000010 \\ 00000010000 \\ 00001000000 \\ 00100000000 \\ 00000100000 \\ 01000000000 \\ 00010000000 \\ 10000000000 \\ \hline
00000000001 \\ 00000001000 \\ 00000000010 \\ 10000000100 \\ 00001000000 \\ 00000010000 \\ 00000100000 \\ 00100000000 \\ 00010000000 \\ 01000000000
\end{array}\right) &
\left(\begin{array}{c}
00000000010 \\ 00001000000 \\ 00000000100 \\ 00000100000 \\ 00000001000 \\ 00000000001 \\ 00000010000 \\ 01000000000 \\ 00010000000 \\ 10000000000 \\ 00100000000 \\ \hline
00100000000 \\ 10000000000 \\ 00010000000 \\ 00000000010 \\ 00001000000 \\ 00000000100 \\ 00000100000 \\ 00000001000 \\ 00000000001 \\ 00000010000
\end{array}\right) &
\left(\begin{array}{c}
00000000100 \\ 00000000001 \\ 00000001000 \\ 00000000010 \\ 00000010000 \\ 10000000000 \\ 00000100000 \\ 00100000000 \\ 00001000000 \\ 01000000000 \\ 00010000000 \\ \hline
00100000000 \\ 00000100000 \\ 01000000000 \\ 00001000000 \\ 00000000100 \\ 00010000000 \\ 00000001000 \\ 00000000001 \\ 00000010000 \\ 00000000010
\end{array}\right)
\end{array}
$$

Figure 5.11: LS-Sig PBX matrices. A 1 in the topmost row and the leftmost column means that the 6th bit of the address is used to compute the 11th bit of the index. A 1 in the bottommost row and the rightmost column means that the 26th bit of the address is used to compute the LSB of the index.

If we use Bloom filters implemented as regular filters, it needs to be ensured that the union of matrices by pairs is of full rank. This way, all hash functions will yield different indexes for a given address. In this chapter, the parallel implementation is used so that there is no need to check the rank of the union of the matrices. Hash functions may yield the same index for a given address because different subarrays are asserted. Even so, the matrices shown in Figure 5.11 have full rank by pairs.

5.2.5. Saving Hardware

LS-Sig can be considered to enable smaller signature sizes, as opposed to improving only the false positive rate. Figure 5.7 shows that Yada and Labyrinth yield the same results if a parallel LS-Sig is used with half the size of generic ones. Intruder and Genome behave similarly, but when the signature is halved, they behave slightly worse. Vacation and Bayes could be in the same group as Yada and Labyrinth, but only when the signature size is not halved from 256 bit downward.

As regards the hash functions, according to the XOR count proposed by Yen et al. [111], $(5, \delta_P)$-LS-Sig reduces the number of XOR gates by about 6.5% with respect to the generic version. On the other hand, PBX can achieve a reduction of up to 80%. However, the design of PBX signatures is subject to prior analysis of the entropy of the workloads. It is to be noted that the area required by hash functions may represent about one-fifth the size of the SRAM for $k = 4$ [92].

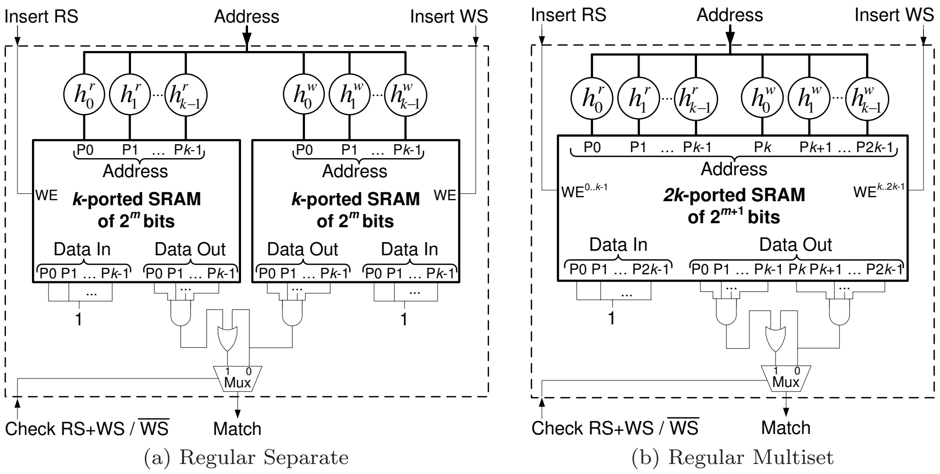
6 Dealing With Asymmetry In Data Sets

In this chapter we present our *MultiSet* and *Reconfigurable Asymmetric* signatures [77, 82, 83] to cope with asymmetry in transactional data sets.

Read and write signatures are usually implemented as two separate, same-sized Bloom filters. In contrast, transactions frequently exhibit read and write sets of uneven cardinality. In addition, both sets are not disjoint, as data can be read and also written. This mismatch between data sets and hardware storage introduces inefficiencies in the use of signatures that have some impact on performance, as, for example, read signatures may populate earlier than write ones, increasing the expected false positive rate.

We present different signature designs as alternative to the common scheme to deal with asymmetry in transactional data sets in an effective way. Basically, two classes of new signatures are analyzed: multiset and reconfigurable asymmetric signatures. The first class uses a single Bloom filter to track both read and write sets. Different alternatives are studied to take advantage of some important properties of data access patterns, like either the significant amount of transactional memory locations that are both read and written, or the locality of reference. The second class uses Bloom filters of reconfigurable size for reads and writes (a static approach was first discussed in [67] where the sensitivity to signature length is analyzed). The main focus of this chapter is a thorough study of these alternative signature designs, including a false positive probability analysis and a complete experimental evaluation comparing the different approaches in terms of performance, and hardware area, time and energy requirements.

The rest of the chapter is organized as follows. Next section introduces the proposed signature schemes, multiset and reconfigurable asymmetric, discussing their concept and implementation, and showing a comparison with the common



(a) Regular Separate (b) Regular Multiset

Figure 6.1: Regular signature implementation schemes.

designs based on separate filters. Section 6.2 shows a statistical analysis of the proposed signatures, determining false positive rates in different contexts. Section 6.3 presents experimental results for the multiset and reconfigurable asymmetric signatures on the GEMS simulator using the STAMP workloads, and compares the performance attained by the different design alternatives. Besides, an analysis of area, time and energy requirements using CACTI is also shown.

6.1. Multiset and Reconfigurable Asymmetric Signatures

This section presents the multiset and reconfigurable assymetric signature proposals and their implementation, both regular and parallel, as alternatives to separate schemes.

6.1.1. Regular Multiset Signatures

Figure 6.1a shows the implementation of a regular separate signature (SEP). It comprises two Bloom filters, one for keeping track of the addresses read by a transaction and the other one for the addresses written. Such filters can be implemented as SRAMs of 2^m bits. When $k > 1$, multi-ported SRAMs are needed to perform the operations in one cycle. Pi, $i \in [0, k-1]$ represents the ports. However, multi-ported memories require more hardware than single-ported ones and signatures must keep both concise and fast. The signature provides four operations: inserting an address into the read set filter (Insert RS), inserting an address into the write set (Insert WS), checking for membership of an address into both read and write sets (Check RS+WS — transactional writes must be checked this way) and checking for membership of an address into the write set (Check $\overline{\text{WS}}$ — for transactional reads). In case of insertion, the Address is mapped into k indexes by hash functions, either h_i^r, $i \in [0, k-1]$, for the read set or h_i^w for the write set. Then, the Insert RS/WS signal enables writing in the SRAM (WE: write enable). In this case WE enables all the ports in the SRAM. The SRAM input ports are set to 1, so the bits indexed by the hash functions are all set to 1. In case of checking for write set membership, the k 1-bit output data ports are ANDed together and Check $\overline{\text{WS}}$ selects the 0 input of the multiplexer which is then connected to Match. For read set and write set checking, both the AND output of the write set and the AND output of the read set are ORed, while Check RS+WS selects the 1 input of the multiplexer which connects it to the Match output.

Regular multiset signatures (MS) join both the read set and the write set filters together in the same filter of twice the size: 2^{m+1} bits. Figure 6.1b depicts how this proposal can be implemented. The read set hash functions and the write set hash functions index the whole SRAM address range. Therefore, $2k$ ports are needed and the Insert RS signal drives the first k WE inputs and the Insert WS signal drives the rest. The duplication of the number of ports of the SRAM leads regular multiset signatures to a quadratic growth of the required area. In order to save in area and also to maintain time-efficiency, parallel signatures are used [14, 92] which do not need multi-ported SRAMs.

6.1.2. Parallel Multiset Signatures

A parallel Bloom filter comprises k arrays of $2^m/k$ bits. Each hash function indexes its own array, so one bit is set into each array on insertion.

Figure 6.2a shows the implementation of parallel separate signatures. Like regular filters, parallel filters can be implemented as SRAMs. However, they use manifold smaller single-ported SRAMs instead of a larger multi-ported one, thus saving in hardware area. Furthermore, parallel Bloom filters have been proved to yield similar or better performance than regular ones [79, 92].

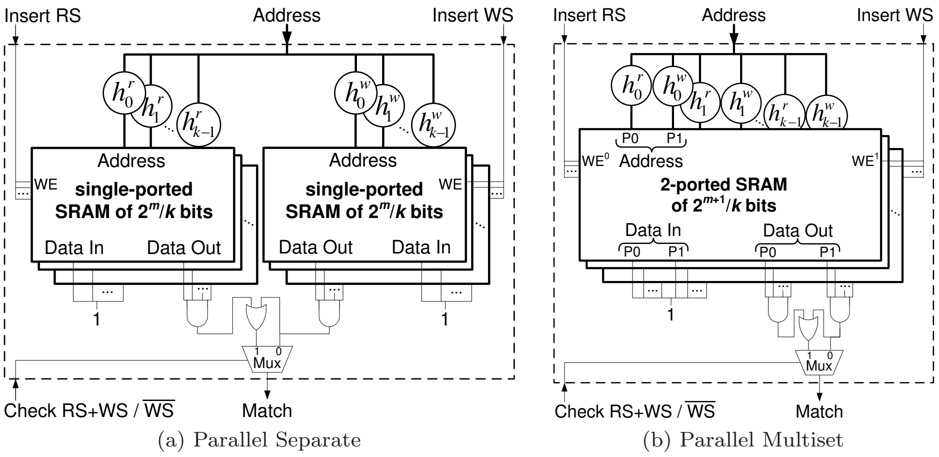

Insert RS
Address
Insert WS
h_0^r h_1^r h_{k-1}^r
h_0^w h_1^w h_{k-1}^w
Address
WE
single-ported SRAM of $2^m/k$ bits
Data In
Data Out
1
Mux
Check RS+WS / $\overline{WS}$
Match
(a) Parallel Separate
P0 P1
WE⁰ Address WE¹
2-ported SRAM of $2^{m+1}/k$ bits
(b) Parallel Multiset

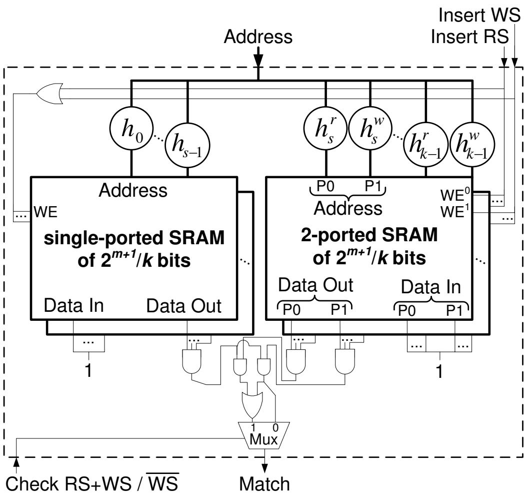
Figure 6.2: Parallel signature implementation schemes.

On the other hand, Figure 6.2b depicts the implementation of the multiset counterpart for the parallel signature. In this case, the SRAM is also partitioned into k smaller arrays but of $2^{m+1}/k$ bits. Now, each SRAM is indexed by two hash functions, one for the read set, h_i^r, and the other one for the write set, h_i^w. Therefore, parallel multiset signatures take over twice more area than parallel SEP signatures, since parallel MS signatures need 2-ported SRAMs whereas parallel SEP signatures use single-ported SRAMs. To reduce the complexity of the multiset scheme, next we propose to keep certain SRAMs single-ported.

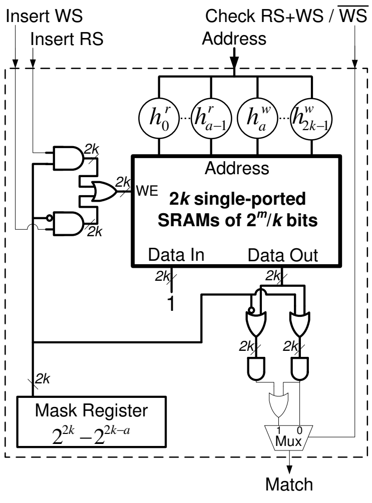
6.1.3. Parallel Multiset Shared Signatures

Several memory locations are read and written inside transactions. Some of them are only read and others are only written but many of them are both read and written. Section 6.3.4 shows that about 30% of locations are both read and written for the workloads tested. In such a case, storing the same address twice is redundant but the filter must be able to discriminate whether the address was only read or also written.

Figure 6.3a shows the proposed solution. The signature is a parallel multiset signature where s SRAMs are single-ported, so they are indexed by only one hash function, $h_0, h_1, ..., h_{s-1}$, with $s \in [0, k]$, i.e. hash functions are said to be *shared* between RS and WS. This way, when inserting an address into the filter, some



(a) Parallel Multiset Shared



(b) Reconfigurable Asymmetric

Figure 6.3: Multiset shared and reconfigurable asymmetric signature implementation schemes.

arrays do not take into account whether the address was either read or written, they simply record one bit representing the address. That is why Insert RS and Insert WS are ORed to drive the WE signal of the single-ported SRAMs. However, the rest of the arrays must continue to discriminate between reads and writes so they are addressed by a read hash function, h^r, and a write hash function, h^w. Consequently, in case of an insertion to the write set, h^w would set a bit in its SRAM. Then, if the same address is subsequently inserted to the read set, a different bit would be set to 1 by h^r in the same SRAM.

On checking the signature, the bits from the single-ported SRAMs, which are the same for the read set and for the write set, are ANDed together and then they are also ANDed to the bits coming out of the double-ported SRAMs, which are different depending on the port: P0 bits correspond to the read set and P1 bits correspond to the write set.

Finally, to find out the value of s, a trade off between hardware requirements and signature performance has to be carried out. On the one hand, if s is set to k, the signature implements k single ported SRAMs. Hence, parallel MS shared signatures require hardware similar to parallel separate signatures but parallel MS signatures will not differentiate between read and written addresses, which could degrade the performance. On the other hand, if s is set to 0, multiset signatures implement k double-ported SRAMs, thus increasing the hardware requirements but maximizing the probabilities of discriminating between read and written addresses. Section 6.3.4 explores every possible scenario.

6.1.4. Reconfigurable Asymmetric Signatures

We propose a reconfigurable asymmetric signature (ASYM) that can be configured at execution time to have a read set larger, the same length or smaller than the write set.

Figure 6.3b depicts the implementation of this signature design. Taking a parallel separate scheme like that of Figure 6.2a as a starting point, the ASYM signature can configure the number of (hash, SRAM)-pairs devoted to each data set instead of being k each. Thus, $2k$ SRAMs of $2^m/k$ bits are needed (depicted by the thick-lined rectangle), each one indexed by its hash function: $h_0^r, h_1^r, ..., h_{a-1}^r$ for the read set and $h_a^w, h_{a+1}^w, ..., h_{2k-1}^w$ for the write set with $a \in [1, 2k-1]$. The parameter a is provided by the Mask Register in terms of a mask whose value can be deducted from the expression $2^{2k} - 2^{2k-a}$, which stands for a number of ones padded with $2k - a$ zeros on the right. For example, on the need for a read set larger than the write set, a can be either 5, 6 or 7 which means that the read set comprises 5, 6 or 7 SRAMs and the write set comprises 3, 2 or 1 SRAMs, respectively. Consequently, the masks would be the next bit words: 11111000, 11111100 and 11111110.

On inserting an address into the asymmetric signature, the SRAMs' WE is selected depending on the Mask Register. For Insert RS, this signal is ANDed together with the mask, because the mask represents the number of SRAMs in the read set. For Insert WS, this signal is ANDed with the inverse of the mask which represents the SRAMs belonging to the write set. Finally, the result of both AND gates is ORed to drive the WE signals of the SRAMs.

On checking for WS membership, the output of the $2k$ SRAMs is bitwise ORed with the mask so that the RS SRAM output bits are ORed with a 1 and they result in a 1 whatever their value. However, the WS SRAM output bits are ORed with the 0 bits of the mask so they stay the same. Then, the $2k$ outputs of the OR gate are ANDed together giving as a result the AND of the WS SRAM output bits as the RS SRAM output bits have been all set to 1. To work out the RS match output, the same procedure is followed but the inverse of the mask is used.

The Mask Register could be loaded with a mask by means of either an instruction in the instruction set architecture or the contention manager of the TM system. In any case, the problem lies in finding the appropriate mask to configure the signature to yield the best performance. This is not a trivial problem, because we might need feedback from the compiler, runtime, TM system or from the behavior of the signature itself. Finding the appropriate mask also depends on whether a static (per-execution) or dynamic (per-transaction) signature configuration is preferred throughout the execution of the application. We do not deal with this problem in this thesis. However, Section 6.3.3 explores several per-execution configurations of the ASYM signature to give more insight in its behavior compared to MS signatures. A heuristic to choose the best per-execution configuration is also proposed.

6.1.5. Hash Functions

Hash functions are implemented as H3 XOR hash functions [12] which comprise a set of XOR gate trees per function. XOR gate trees do not require significant area and, moreover, they can be replaced by a single line of XOR gates by using PBX hashing [111].

However, the area of the hashing logic depends on the number of hash functions k and the number of bits required to address the SRAMs, which depends on whether the signature is implemented as regular, parallel, separate or multiset. Also, half of the address bits are used per bit of the hash function on average [111], so the number of 2 fan-in XOR gates needed by an XOR tree that computes one hash bit is $b = \lceil \frac{address\ length}{2} \rceil - 1$. Then, the expressions which determine the number of XOR gates for the different signatures schemes are the following:

- *Regular separate signatures*: as Figure 6.1a shows, regular separate signatures comprises $2k$ hash functions, k for the read set and k for the write set. However, as read set is separated from the write set they can use the same hash functions. Finally, the number of bits they need to index their arrays is m, then:
$$\#XOR = b \cdot m \cdot k \tag{6.1}$$
- *Regular multiset signatures*: the multiset signature joins the RS and WS SRAMs together so their hash functions need one more output bit to index the SRAM, $m + 1$, while the hash functions must be different each other:
$$\#XOR = b \cdot (m + 1) \cdot 2k \tag{6.2}$$

- *Parallel separate signatures*: depicted in Figure 6.2a, this scheme divides the two SRAMs into k smaller SRAMs of $\frac{2^m}{k}$ bits. The number of hash functions still is k but the index bits are $m - log_2 k$ in this case:

$$\#XOR = b \cdot (m - log_2 k) \cdot k \tag{6.3}$$

- *Parallel multiset signatures*: the parallel multiset signature is like its separate counterpart but requires 1 more bit per hash function and different hash functions for read set and write set:

$$\#XOR = b \cdot (m + 1 - log_2 k) \cdot 2k \tag{6.4}$$

- *Parallel multiset shared signatures*: shown in Figure 6.3a, their hash functions yield indexes of the same length as those of multiset signatures. However, the number of hash functions changes in this case, because the s single-ported SRAMs are only indexed by one hash function each, so the total number of hash functions is $2k - s$:

$$\#XOR = b \cdot (m + 1 - log_2 k) \cdot (2k - s) \tag{6.5}$$

- *Asymmetric signatures*: these signatures are similar to parallel separate signatures but, in this case, the number of different functions is $2k - 1$, because a can range from 1 to $2k - 1$.

$$\#XOR = b \cdot (m - log_2 k) \cdot (2k - 1) \tag{6.6}$$

Next, an example for real parameters is shown. For an address of 26 bits (32 bits - 6 bits of line address), $m = 10$ and $k = 4$, 480 XOR gates are needed for regular separate signatures. The multiset counterpart results in 1056 gates. Parallel separate signatures need 384 XOR gates whereas the parallel multiset one needs 864. The asymmetric signature needs 672. Note that multiset schemes need one more bit per hash index and twice the number of hash functions of separate signatures, because the arrays are double sized and share the filter. However, the multiset shared scheme lowers the XOR gate requirements by lowering the number of hash functions. For example, the multiset shared signature with $s = 1$ needs 756 XOR gates. With $s = 2$ only 648 gates, and with $s = 3$ it just needs 540 XOR gates, quite close to those needed by parallel separate signatures.

The expressions above provide an upper bound for the number of XOR gates required by the hash functions. They use half of the address bits per hash bit so the hash functions share many bit pairs of the address and hence, they can also share XOR gates. In fact, in Section 6.3.6, we found that the real number of XOR gates is lower than that given by Expressions 6.1 to 6.6.

6.2. False Positive Analysis

In Section 2.3.1, we formulate the false positive expression of a Bloom filter. It can be computed as a function of its occupancy, that is, the number of bits asserted, and the number of bits to be checked in a query. For an M-bit filter, if bits are asserted equiprobably by a hash function, the probability of a given bit being a 1 is $\frac{1}{M}$. Thus, $1 - \frac{1}{M}$ is the probability of a bit being 0. If *occ* is the occupancy of the filter, that is, *occ* bits have been already asserted, the probability of a bit still being 0 is $(1-\frac{1}{M})^{occ}$, and hence, if *nchecks* bits are going to be checked, the probability of getting a positive, i.e. all checked bits equal to 1, is:

$$p_{\mathrm{P}}(M, occ, nchecks) = \left(1 - \left(1 - \frac{1}{M}\right)^{occ}\right)^{nchecks}. \tag{6.7}$$

A common assumption is that the probability of false positive is approximately equal to that of getting a positive. The reason for that is Bayes' theorem: $Pr(\text{Positive} \bigcap \text{False Positive}) = Pr(\text{False Positive} \mid \text{Positive})Pr(\text{Positive})$. Provided that the number of elements inserted into the filter is a small fraction of the total possible elements whose query is positive, the conditional probability is $Pr(\text{False Positive} \mid \text{Positives}) \approx 1$. In this way:

$$\begin{aligned} &Pr(\text{False Positive}) = \\ &= Pr(\text{False Positive} \mid \text{Positive})Pr(\text{Positive}) \approx \\ &\approx Pr(\text{Positive}) \end{aligned} \tag{6.8}$$

A more usual form of Expression 6.7 for a single filter of M bits, after inserting a sequence of q elements using k hash functions, is:

$$p_{\mathrm{FP}}(M, q, k) = p_{\mathrm{P}}(M, qk, k) = \left(1 - \left(1 - \frac{1}{M}\right)^{qk}\right)^{k}. \tag{6.9}$$

The last expression assumes that elements map into different bits during insertion. So, the occupancy is the number of inserted elements, q, by the number of hash functions, k, and each query checks k bits.

The goal in this section is to evaluate the probability of false positives for an asymmetric/multiset configuration, as the one shown in Figure 6.4, where the Bloom filter array is split into three sections: one exclusively for RS (read section), other exclusively for WS (write section) and the last one for $RS \cup WS$ (multiset section). Let M be the total number of bits of the array. Then, M can

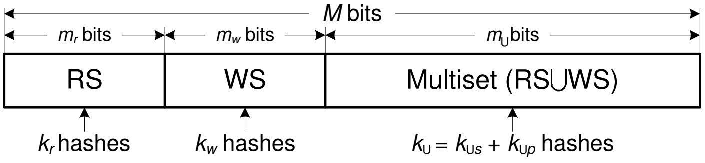


Figure 6.4: Asymmetric/multiset filter under analysis.

be broken down into m_r bits for the read section, m_w bits for the write section and $m_\cup$ bits for the multiset one ($M = m_r + m_w + m_\cup$). Also, the number of hash functions used in each section is defined as: k_r, k_w and $k_\cup$, respectively. The hash functions used for the multiset section, $k_\cup$, are in turn defined as $k_\cup = k_{\cup s} + k_{\cup p}$, where $k_{\cup s}$ functions are shared between RS and WS providing information about insertion only, not whether the address was read or written, and $k_{\cup p}$, though, are private functions which can differentiate reads from writes.

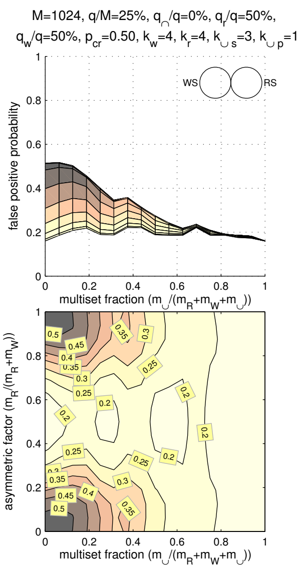
Consider a sequence of $q = Card(RS \cup WS)$ addresses to be inserted into the filter, where $q_r = Card(RS)$, $q_w = Card(WS)$, $q_\cap = Card(RS \cap WS)$ and consequently $q = q_r + q_w - q_\cap$. The false positive rate for the multiset section is given by the following expression, as an address which has been both read and written will assert $k_{\cup s}$ bits ($k_{\cup s}$ hash functions are shared), and $2k_{\cup p}$ bits ($k_{\cup p}$ hash functions are private for read and write sets):

$$
\begin{aligned}
&p^{\cup}_{\mathrm{FP}}(m_\cup, q_r, q_w, q_\cup, k_{\cup s}, k_{\cup p}) = \\
&\quad p_{\mathrm{P}}(m_\cup, k_{\cup s}(q_r + q_w - q_\cap) + k_{\cup p}(q_r + q_w), k_{\cup s} + k_{\cup p}).
\end{aligned}
\tag{6.10}
$$

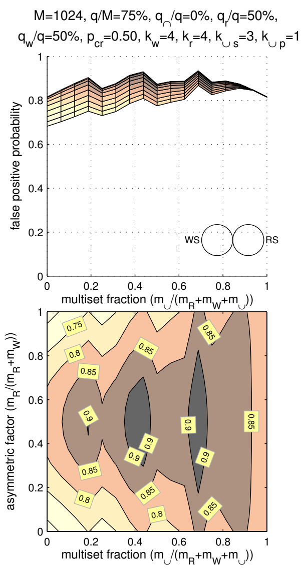
According to Expression 6.8 the probability of getting a false positive on checking a read involves getting a positive both in the read section and the multiset section of the filter in Figure 6.4. The argument is analogous for writes. Thus, the mathematical expectation of getting a false positive can be obtained considering the probabilities of checking a read (p_{cr}) and a write (p_{cw}). This way:

$$
\begin{aligned}
&E_{\mathrm{FP}}(m_r, m_w, m_\cup, q_r, q_w, q_\cap, k_r, k_w, k_{\cup s}, k_{\cup p}, p_{cr}, p_{cw}) = \\
&\quad p^{\cup}_{\mathrm{FP}}(m_\cup, q_r, q_w, q_\cap, k_{\cup s}, k_{\cup p}) \times \\
&\quad (p_{cr} p_{\mathrm{FP}}(m_r, q_r, k_r) + p_{cw} p_{\mathrm{FP}}(m_w, q_w, k_w)).
\end{aligned}
\tag{6.11}
$$

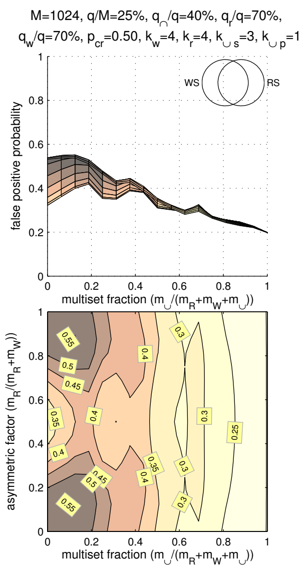
Note that p_{cr} and p_{cw} are conditioned by the TM system. In our TM simulation environment, about 94% of checks involved both checking read and write



(a)



(b)

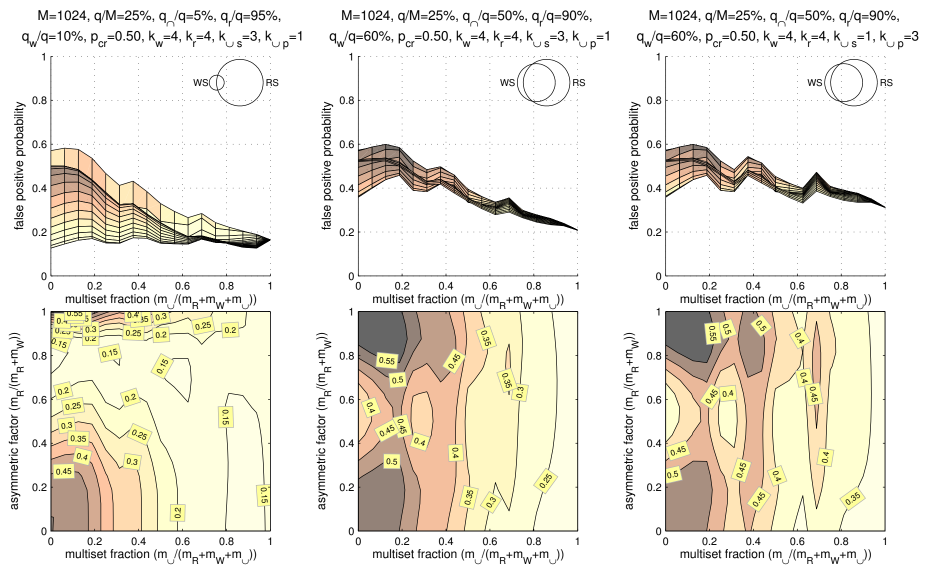


(c)

Figure 6.5: Side view and contour plot of the surface formed by the expected value of the false positive probability according to Expression 6.11. Part 1 of 3.

signatures, whereas the rest were write checks only (see Section 6.3.4). This is due to invalidations, replacements and L2 cache misses which are frequent events in contended codes with large transactions, exacerbated by LogTM's cacheable logs. Such events needs both read and write filters to be checked to ensure isolation in the virtualized TM system [110]. Therefore, we can assume that $p_{cr} = p_{cw} = \frac{1}{2}$.

Several situations of interest have been evaluated in Figures 6.5, 6.6 and 6.7. Plots represent Expression 6.11 considering two variables for the filter in Figure 6.4: the asymmetry between the read and write sections (labeled as asymmetric factor in plots), and the fraction of the total filter taken up by the multiset section (labeled as multiset fraction). Both variables range from 0 to 1. The lower part corresponds to a contour plot of the expected false positive probability, whereas a side view of the surface is depicted in the upper part, showing more clearly its minimum values. The cardinality of RS and WS is sketched in a Venn's diagram. It is to be noted that k_r and k_w values shown in each plot corresponds to the case of a symmetric separate configuration ($m_r = m_w$ and $m_{\cup} = 0$). As well, $k_{\cup s}$ and $k_{\cup p}$ values shown in each plot corresponds to a full



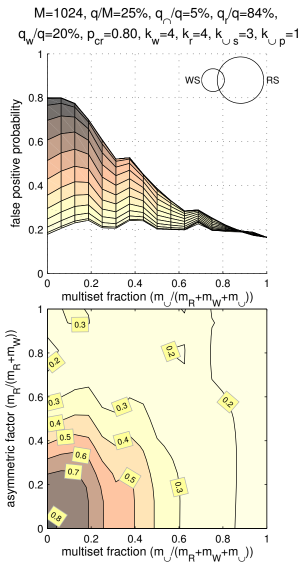
(a) (b) (c)

Figure 6.6: Side view and contour plot of the surface formed by the expected value of the false positive probability according to Expression 6.11. Part 2 of 3.

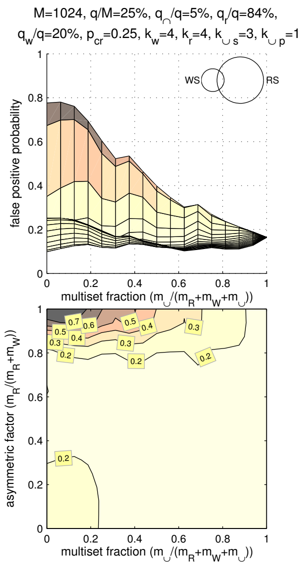
multiset configuration ($M = m_\cup$). For the rest of the evaluated points, k parameters were interpolated proportionally to their associated filter sections (m_r, m_w and $m_\cup$) and rounded to their ceiling values.

Figure 6.5a, 6.5b and 6.5c corresponds to a symmetrical insertion pattern, where the cardinality of RS and WS are identical. When the occupancy is low (Figure 6.5a) the symmetrical separate configuration and the full multiset get similar false positive rates. Nevertheless, the situation changes when the filter is more populated (Figure 6.5b). In this case, asymmetric separate configurations exhibit better false positive rates. A heuristic occupancy threshold of $q/M \approx 2/3$ were found, which separates these two scenarios. Figure 6.5c shows the effect of having non null $RS \cap WS$, which increases the number of insertions, because there are memory locations that are inserted as both read and written. Thus, higher occupancy is expected for non multiset filters, and consequently, more false positives.

Figure 6.6a illustrates an asymmetric insertion pattern where $Card(RS) > Card(WS)$. In this case, several configurations lead to the minimum false positive



(a)



(b)

Figure 6.7: Side view and contour plot of the surface formed by the expected value of the false positive probability according to Expression 6.11. Part 3 of 3.

probability. Both full multiset and a separate asymmetric solution can be chosen.

Scenarios of Figures 6.6b and 6.6c, where almost $WS \subset RS$, give advantage to the full multiset configuration versus having RS and WS completely separated. Nevertheless, as $k_{\cup p}$ grows with respect to $k_{\cup s}$, the advantage gets smaller, because addresses that are both read and written assert more bits in the multiset part, as can be observed in Figure 6.6c.

Finally, Figures 6.7a and 6.7b show two completely asymmetric situations breaking the preceding assumption that $p_{cr} = \frac{1}{2}$. In Figure 6.7a, both the insertion and checking patterns are biased to reads ($Card(RS) > Card(WS)$ and RS is the most frequently checked). Here, a separate asymmetric configuration gets a slightly lower false positive probability than the full multiset. In Figure 6.7b, $Card(RS) > Card(WS)$ as well, but WS is the most frequently checked set. This last situation may happen if the probability of checking the write set is much larger than that of checking the read set. Here, the full multiset configuration has no advantages over the separate.

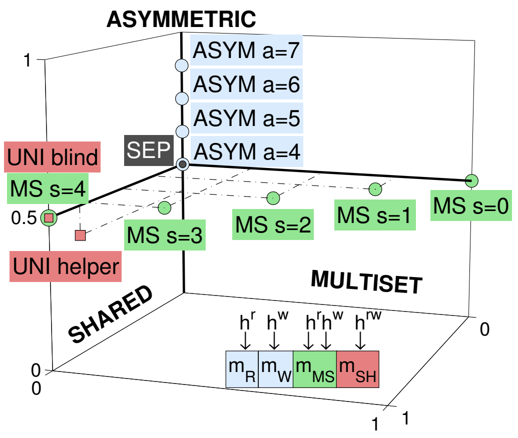


Figure 6.8: Explored solutions.

Next section evaluates several implementations of the analyzed signatures. Figure 6.8 shows the explored solutions in terms of three dimensions: read asymmetry (m_R to m_W ratio), hash sharing ($k_{\cup s}$ fraction of $m_\cup$; m_{SH} in Figure 6.8) and multiset ($k_{\cup p}$ fraction of $m_\cup$; m_{MS} in Figure 6.8). Our proposed signatures are marked with circles: four ASYM and five MS schemes. The SEP scheme, symmetric and separate, is equivalent to ASYM $a = 4$. Unified (UNI) blind and helper signatures from Choi et al. [21] are also shown, marked with squares. Note the equivalence between MS $s = 4$ and UNI blind schemes. The UNI helper signature remains in the shared plane but slightly shifts on the asymmetric axis due to its helper write register.

6.3. Experimental Evaluation

In this section, methodology (Section 6.3.1), experimental results (Sections 6.3.2, 6.3.3, 6.3.4 and 6.3.5) and hardware requirements (Section 6.3.6) are described.

6.3.1. Methodology

The methodology outlined in Chapter 3 was used to evaluate the signatures schemes described in Section 6.1. The Ruby module that implements the TM system was modified to include all proposed signature schemes described in Sec-

Table 6.1: Workload transactional characteristics. (Data set sizes in the right-most columns are reported in cache blocks)

Bench	**#xact**	**Time in xact**	***avg* $\|RS\|$**	***avg* $\|WS\|$**	***max* $\|RS\|$**	***max* $\|WS\|$**	$\frac{avg\|RS\|}{avg\|WS\|}$
Bayes	523	94%	76.9	40.9	2067	1613	1.88
Genome	30304	86%	12.1	4.2	400	156	2.88
Intruder	12123	96%	19.1	2.5	267	20	7.64
Kmeans	1380	6%	99.7	48.5	134	65	2.06
Labyrinth	158	100%	76.5	62.9	278	257	1.22
SSCA2	47295	19%	2.9	1.9	3	2	1.53
Vacation	24722	97%	19.7	3.6	90	30	5.47
Yada	5384	100%	62.7	38.4	776	510	1.63

tion 6.1.

Perfect signatures were used as a reference, because they do not yield false positives. Filter size ranged from 64 bits, which matched the word length in SPARC architecture, to 8K bits length, which matched the performance of perfect signatures for the simulated benchmarks. All filters used 4 hash functions of the H3 family and the same H3 matrices of Ruby.

The proposed signature schemes were experimentally evaluated using all the codes of the STAMP suite introduced in Section 3.2. The benchmarks were run with the input parameters shown in Table 5.3. Table 6.1 shows main transactional characteristics of the benchmarks. “#xact” is the number of committed transactions. Column “Time in xact” lists the percentage of execution cycles of the benchmarks spent inside transactions. The last columns show the average and the maximum values of RS and WS size distributions in cache blocks as well as the ratio.

6.3.2. Regular and Parallel Multiset Signatures Results

Figure 6.9 shows the results obtained for regular and parallel separate and multiset signatures. The y axis represents the time in cycles which was normalized to that of perfect signatures. Solid lines depict regular signatures and dashed lines depicts parallel ones. The x axis represents the size of the filter. For example, a 64bit value means that separate signatures use two 64bit filters, one for the RS and one for the WS, while multiset signatures use only one 128bit filter. According to their behavior, benchmarks can be classified into three different groups:

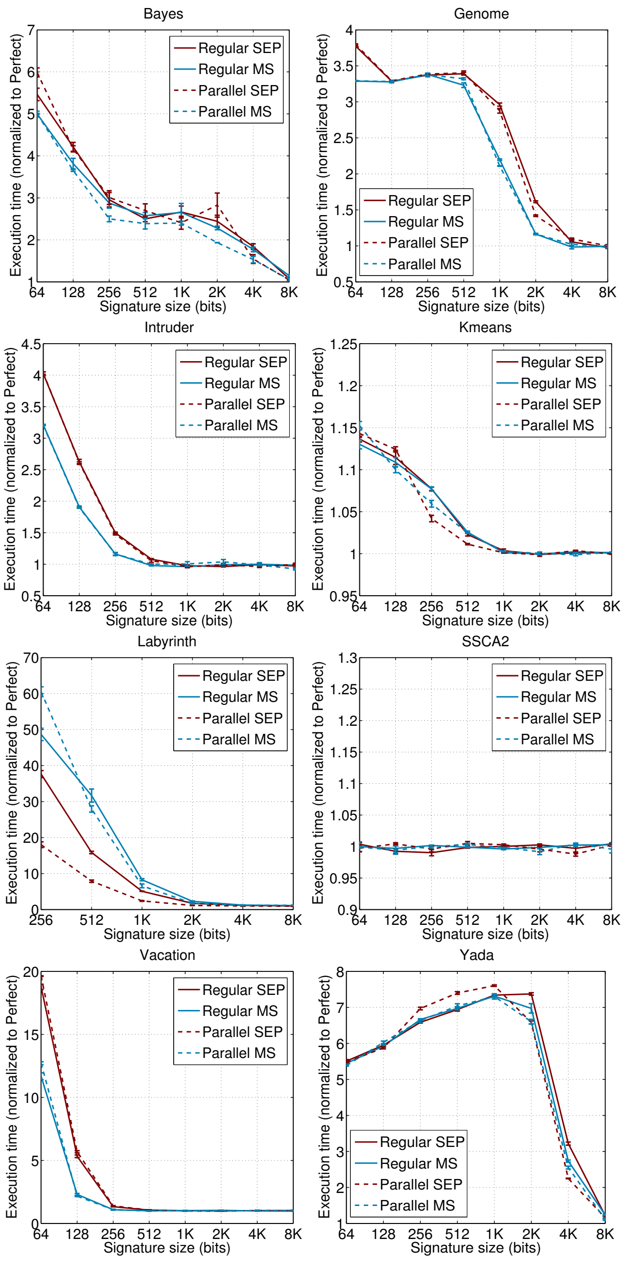


Figure 6.9: Execution time normalized to perfect signatures comparing regular and parallel separate signatures (SEP) to regular and parallel multiset ones (MS).

1. SSCA2: This benchmark is not signature dependent, because of its small transactions, the smallest of the whole suite as Table 6.1 shows. In addition, it spends most of the time outside transactions.

2. Bayes, Genome, Intruder, Vacation and Yada: These five workloads behave better when using multiset signatures instead of separate ones.

   Bayes and Yada reap a slight improvement of their execution time for certain signature sizes, about 1.2× for Bayes with parallel small signatures and 1.2× for Yada with regular large ones. These benchmarks show large transactions that introduce *cross false positives*. Cross false positives appear in multiset signatures as filter fills. For example, a transaction that inserts only reads in its signature could yield a cross false positive, because of filter occupancy, if a test for a write hits the signature. Figure 6.10 shows the average false positive percentage for regular SEP and MS signatures. High cross false positive percentages can be appreciated for Bayes, Genome, Intruder, Vacation and Yada but the overall false positive percentage is lower than that for SEP signatures. Notice that MS signatures equalizes the number of read set and write set false positives.

   On the other hand, Genome, Intruder and Vacation perform better using multiset signatures. Genome is 1.4× faster with 1Kbit and 2Kbit filters. Intruder also exhibits about 1.4× speedup from 256bit filter downwards, and up to 2.5× of speedup is achieved for Vacation. These three benchmarks show relatively small transactions on average (see Table 6.1) and get not too much affected by cross false positives.

3. Kmeans and Labyrinth: Multiset signatures do not properly work with these workloads. Regular MS signatures perform like regular SEP ones for Kmeans but parallel MS signatures perform worse for some filter sizes. For Labyrinth, MS signatures perform much worse than SEP filters specially for parallel ones. Labyrinth's transactions are large on average and fill the filter beyond the 2/3 threshold (see Section 6.2) introducing many cross false positives. Figure 6.10 shows that, in this case, cross false positives translates into a higher overall false positive percentage than that for SEP signatures. Next sections propose certain configuration enhancements that will ameliorate filter occupancy.

Parallel signatures perform similar than regular ones in most cases, as shown in Figure 6.9, and require much less area (see Section 6.3.6). Therefore, subsequent optimizations are explored using the parallel scheme.

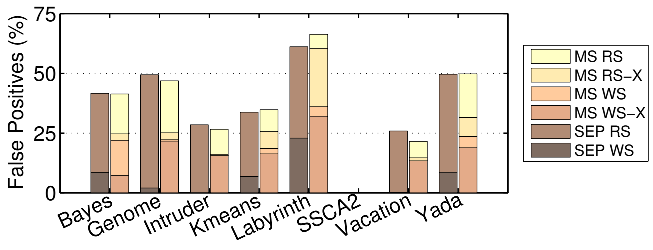


Figure 6.10: Average false positive percentage for regular SEP and MS signatures broken down into RS, WS and cross false positives (RS-X, WS-X).

6.3.3. Reconfigurable Asymmetric Signatures Results

In this section a comparison between parallel separate, parallel multiset and parallel reconfigurable asymmetric signatures is conducted.

Reconfigurable asymmetric signatures, as seen in Section 6.1.4, can be configured to have different read set and write set sizes. The reconfiguration could be performed dynamically at run time. However, in this case some static per-execution configurations are tested as dynamic reconfiguration would need feedback from different parts of the TM system, runtime or compiler, which has not been addressed in this thesis. Therefore, three asymmetric configurations are shown in Figure 6.11: $a=5$ which means that the read set comprises 5 SRAMs and the write set 3 SRAMs, $a=6$ which devotes 6 SRAMs to the RS and 2 to the WS, and $a=7$ with 7 RS SRAMs and just 1 WS SRAM. Configurations with larger WS than RS are not taken into account, because benchmark transactional characteristics in Table 6.1 showed that the average RS size of transactions is always larger than or similar to the average WS size for the tested codes.

As shown in Figure 6.11, the best asymmetric configuration for each benchmark behaves worse than or similar to the multiset signatures, except for the benchmarks which already behaved badly with multiset signatures, i.e. Kmeans and Labyrinth.

It is to be noted that the best asymmetric configuration could be chosen by studying the average ratio of the data sets of the benchmarks. Last column of Table 6.1 shows the ratio between the average RS size and the average WS size of transactions. Bayes has a data set ratio of 1.88 and it behaves better with $a=5$, as $a=5$ involves a 1.66 filter ratio which is closer to 1.88 than $a=6$ with a ratio of 3 and $a=7$ with a ratio of 7. Genome exhibits a ratio of 2.88 wich

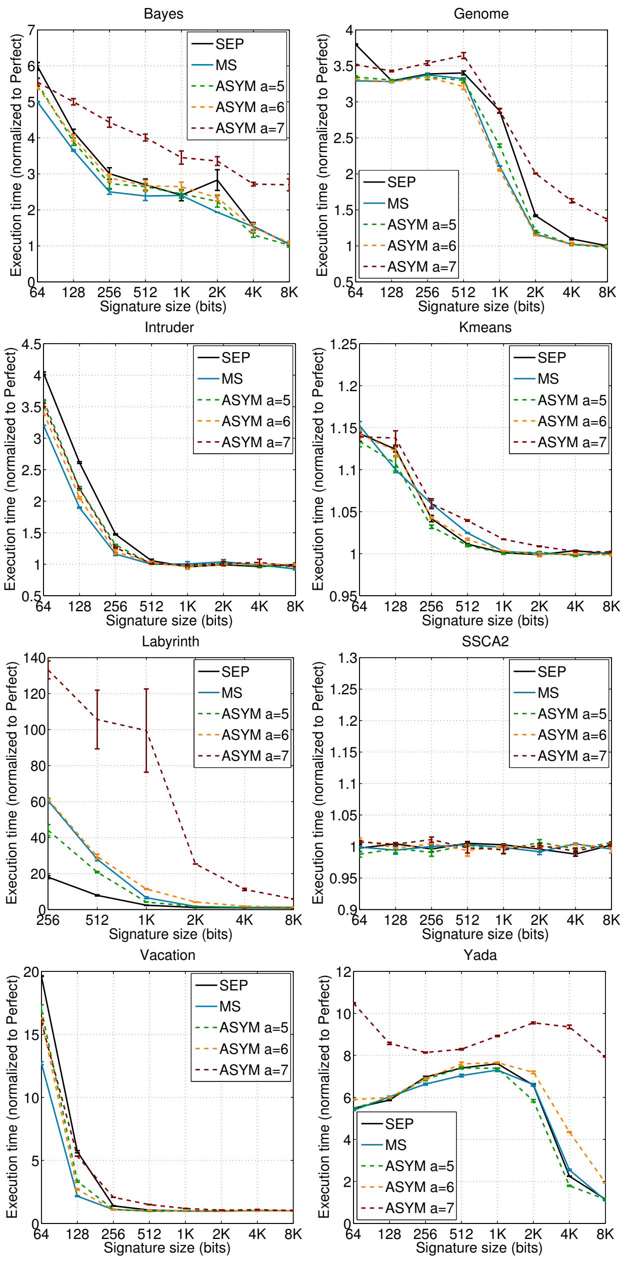


Figure 6.11: Execution time normalized to perfect signatures comparing parallel separate, multiset and asymmetric signatures varying parameter a.

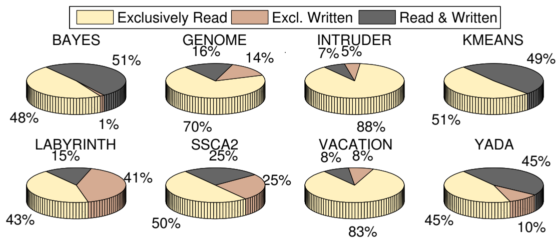


Figure 6.12: Percentage of addresses exclusively read, written and both read and written inside transactions.

almost matches the ratio of 3 for $a=6$ with which it performs better than other asymmetric configurations. Kmeans is 2.06 which is closest to $a=5$ and it yields the best results. For Labyrinth, the parallel separate version (SEP) shows the best results as it actually is a reconfigurable asymmetric signature with $a = 4$ and a ratio of 1. As seen in Table 6.1, Labyrinth has a 1.22 RS to WS ratio, so it is closest to $a = 4$ than to $a=5$. Yada shows a ratio of 1.63. This would have lead to choose $a=5$ as configuration value and results shows that it is the best choice in this case. However, Intruder and Vacation fail to perform the best with the configuration parameter suggested by their RS to WS ratio. Intruder shows a 7.64 ratio closest to $a=7$ but in this case the best asymmetric configuration is $a=6$. And, Vacation's ratio is 5.47 but the best configuration is $a=6$ as well.

Therefore, the RS to WS ratio is a heuristic to choose the configuration for ASYM signatures but more feedback from other parameters is needed to assure best results. As ASYM signatures are not a general solution if we lack effective mechanisms to dynamically reconfigure them on a per-transaction basis, they were taken aside to gain more insight in MS signatures in next sections.

6.3.4. Parallel Multiset Shared Signatures Results

Parallel multiset shared signatures were described in Section 6.1.3. The motivation behind such a signature comes from Figure 6.12, which shows the percentage of addresses that are either exclusively read, exclusively written or both read and written inside transactions for each benchmark. For example, Bayes and Kmeans exhibit close to 100% of written addresses that were also read. Overall, about 30% of total memory locations accessed by each benchmark are both read and written. As the percentage of addresses both read and written inside transactions is significant, next step is figuring out the number of filters that could

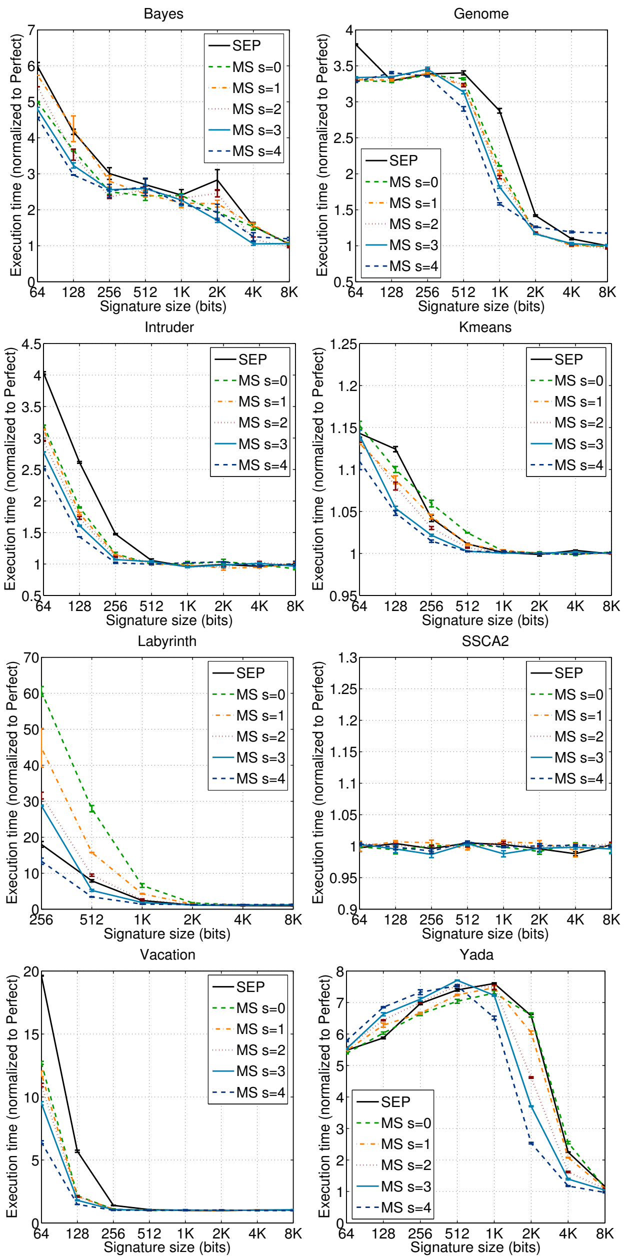


Figure 6.13: Execution time normalized to perfect signatures comparing parallel separate and multiset signatures varying parameter s.

be implemented as single-ported SRAMs in multiset signatures without losing performance. For that purpose, we conducted experiments where parameter s ranged from 0, which is equivalent to having a parallel multiset signature, to 4 functions, which means that every insertion into the read set is also an insertion into the write set and vice versa.

Figure 6.13 shows the execution time for parallel multiset shared signatures. As read and write sets hash functions are shared the results get better for all the benchmarks. In fact, MS $s=4$ reaps the best results for every workload except for Bayes and Genome, whose execution time slowed down about 1.25× with respect to parallel filters for 8Kbit signatures. Therefore, parallel multiset $s=3$ signatures are conservatively chosen instead of $s=4$ for the sake of generality, since they perform equal or better than parallel separate filters for all signature sizes, while $s=4$ signatures perform better than $s=3$ for small signatures but worse for large signatures.

Next, we study why execution is not harmed to a large extent when using multiset shared signature schemes, despite of the detection of read-read dependencies as conflicts due to signature sharing.

In order to obtain the percentage of false positives due to the use of multiset shared signatures with $s=4$, we devise a *multiset shared perfect signature* implemented as one set data structure storing addresses regardless of they are read or written. Such perfect structure does not yield false positives due to aliasing but can yield false positives due to set sharing. A perfect separate filter was used for comparison to get the number of false positives due to signature sharing. So, having one set to store both read and written addresses issued by a transaction, set sharing false positives show up in two situations: (i) when a read address, A, is inserted in the signature and a Check WS for A, issued by a read of A from another transaction, is to be performed on such a signature. In this case, false positives become false read-read conflicts; and (ii) when a written address, B, is inserted in the signature and a Check RS+WS for B is to be carried out. In this case, false positives arise in the RS filter but they do not become false conflicts since B was actually written, so true conflicts are detected.

Figure 6.14 shows the percentage of WS signature checks (both from Check WSs and Check RS+WSs) broken down into negative checks and positive matches (both true and false). We can see that the percentage of false positives is substantial. However, the percentage of total Check WSs, which are the checks whose false positives become false read-read conflicts, is very low as depicted in Figure 6.15. Such figure shows the total number of RS+WS filter checks and WS filter checks for each benchmark. It also shows the average percentage which is about 6% for

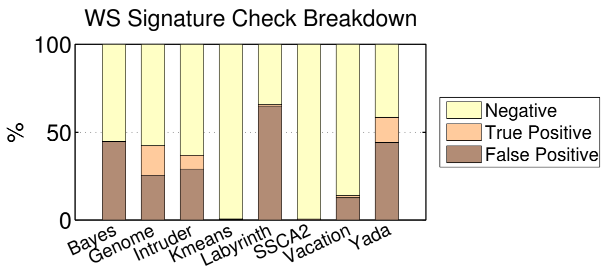


Figure 6.14: Breakdown of WS signature checks into negatives and positive matches (true and false) for all the benchmarks using multiset shared perfect signatures, which implies that false positives are due to read-read dependencies.

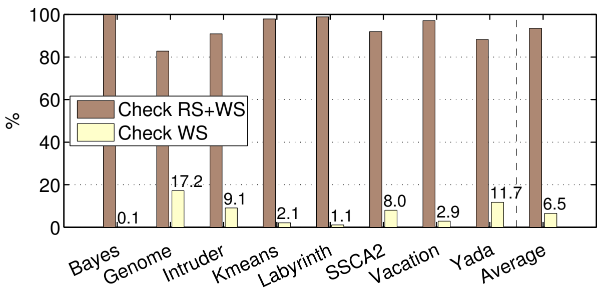


Figure 6.15: Percentage of Check RS+WS and Check WS in the execution of each benchmark using perfect signatures. Average percentage is also shown.

Check WSs. So, from the large amount of false positives in Figure 6.14, only the 6% on average provokes read-read false conflicts. Thus, multiset shared signature schemes do not get too much affected by read-read dependencies.

6.3.5. Enhancing Multiset Signatures with Locality-Sensitive Hashing

In this section, locality-sensitive hashing proposed in Chapter 5 was used to enhance $s=3$ parallel multiset signatures discussed in the preceding section.

Locality-sensitive hashing takes advantage of locality of reference, which is usually exhibited by applications to a greater or lesser extent, to store a set of addresses more concisely. In a Bloom filter with locality-sensitive hash functions,

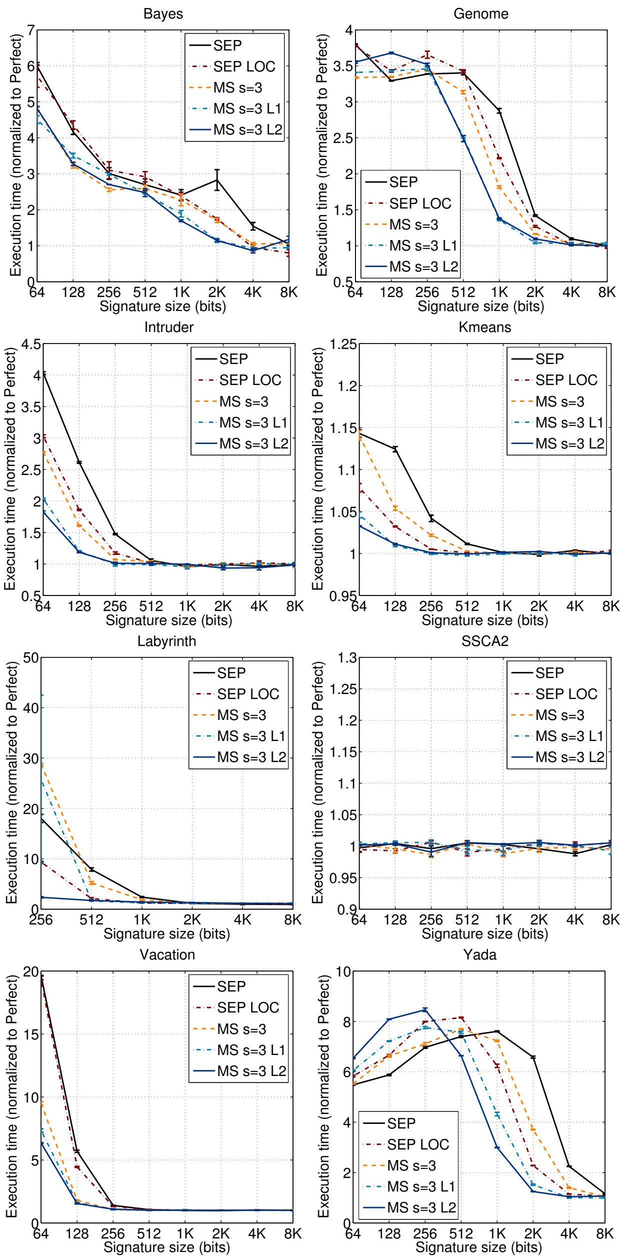


Figure 6.16: Execution time normalized to perfect signatures comparing parallel separate, locality separate and locality multiset $s = 3$ signatures (L1 and L2).

nearby locations assert non-disjoint bits into the bit array saving occupancy. Locality hash functions operate as follows. An address maps into k different indexes, but only one hash index is different between two contiguous addresses. Addresses with distance two have two different indexes. Addresses with distance greater than $2^{k-1} - 1$ may have no hashing outputs in common. That is, one hash operates as a generic H3 hash, but the others take advantage of locality with different granularity.

Figure 6.16 shows the results of enhancing parallel multiset $s=3$ signatures with locality-sensitive hashing. Two different combinations are shown:

- L1: MS $s=3$ signatures share h_0, h_1 and h_2 hash functions while h_3^r and h_3^w functions remain separate (see Figure 6.3a) and assert different bits in the same filter, some bits for the read set and some others for the write set. In the first locality scheme (L1), h_3^r and h_3^w take advantage of locality with maximum granularity, h_2 and h_1 dwindle the granularity and h_0 behave as a generic H3 function. This way, separate functions discriminating locations of read set, h_3^r, from locations of write set, h_3^w, assert less bits in its filter, thus reducing the false positive rate but failing to discriminate locations read from nearby located writes.

- L2: In this case, h_0 is the function which takes advantage of locality with maximum granularity, while h_3^r and h_3^w behave as generic H3 functions, i.e. the filter which do not share the hash functions stays the same as in $s=3$ configuration, thus discriminating between locations read and written, and the other filters get the locality improvement.

As Figure 6.16 shows, results for L1 scheme are practically the same as those for L2 for every benchmark except for Labyrinth, Genome and Yada. Labyrinth behaves better with L2 for small signatures, but Genome and Yada get slightly worse results. MS shared locality signatures outperform parallel and locality separate ones in most cases. Figure 6.17 shows the average speedup per benchmark of MS signatures over parallel separate signatures. It is to be noted that average speedup ranges from 15% of MS $s = 3$ to 47% of MS $s = 3$ L2. Since such values can change if we choose different signature sizes to calculate the average speedup (i.e. we could take signature sizes greater than 8Kb, with speedups of 1, which would cause a drop in the average, or, we could obviate 64bit and 128bit signature sizes as they might be too small for certain workloads), Figure 6.18 shows the average speedup of all benchmarks by signature size. We can see that the MS $s = 0$ configuration speeds the execution up around 20% for 64 and 128bit signature sizes, but the subsequent sizes are affected by the bad results obtained

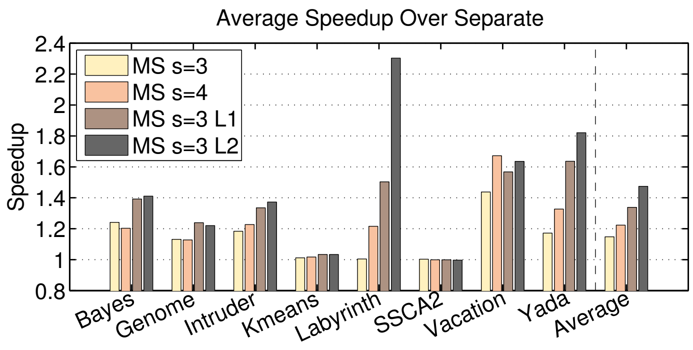


Figure 6.17: Average speedup, per benchmark, of parallel MS $s=3$, MS $s=4$, MS $s=3$ L1 and MS $s=3$ L2 signatures over parallel separate signatures.

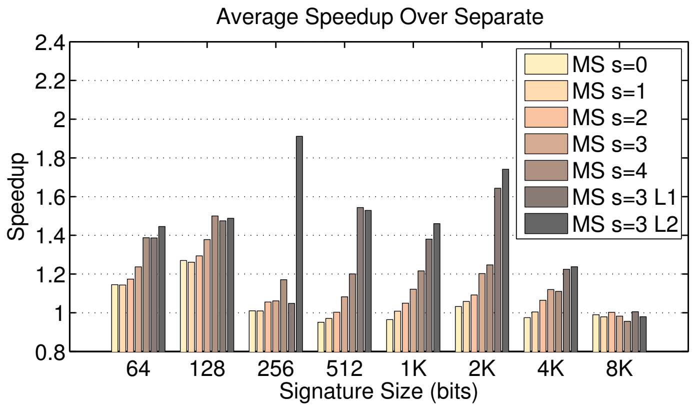


Figure 6.18: Average speedup, per signature size, of parallel MS $s \in [0,4]$, MS $s=3$ L1 and MS $s=3$ L2 signatures over parallel separate signatures.

by Labyrinth, that makes the average speedup fall down close to the separate signature. As parameter s increases, the effect of Labyrinth vanishes until getting up to 99% average speedup for MS $s = 3$ L2 and 256bit signature size and about 75% speedup for 2Kbit signature size. Notice the effect of read-read dependencies of MS $s = 4$ for 8Kbit signature size. The average speedup falls to 0.95.

6.3.6. Hardware Implementation

This section deals with area, time and energy requirements for the proposed signatures.

Table 6.3 shows the results gathered for different filter sizes. “Filter size” row stands for the size of one set filter, i.e. 4Kbit means two filters of 4Kbit (for RS and WS) for separate signatures and one filter of 8Kbit for multiset ones. CACTI 6.5 [72, 109] is used to model the SRAMs using the 65nm process. Due to limitations of CACTI, SRAMs under 1Kbit are not considered. All SRAMs have separate read/write ports, which are dual-ended, meaning that two lines are required per bitline. Output/input bus width is set to one and CACTI’s optimization function searches for the best partition of the cell array depending on time, power and area efficiency.

Concerning SRAMs, regular SEP signatures have two 4-ported SRAMs, one for the RS and one for the WS (see Section 6.1). Regular MS signatures have one 8-ported SRAM and they are three times as large as separate ones due to port increase. Parallel SEP and Asymmetric signatures comprise eight single-ported SRAMs, so they are more concise than regular ones, specifically, 6.5 times on average. Parallel MS signatures have four double-ported SRAMs and they are more than twice as large as parallel SEP signatures, because of the double-ported SRAMs. Parallel MS $s=3$ signatures have three single-ported SRAMs and only one double-ported SRAM lowering the area up to 1.2 times the area of parallel SEP signatures. Finally, an alternative hardware implementation to parallel MS $s=3$ is introduced. We have called it parallel $s=3$, since the multiset part is dropped, and it comprises three single-ported SRAMs for the shared part and two single-ported SRAMs of half the size instead of one double-ported SRAM, which was the former MS part and now is a separate part. This scheme reaps execution time similar to parallel MS $s=3$ (Figure 6.19 shows the behaviour of parallel $s=3$ compared to parallel MS $s=3$ without LS-hashing) but less hardware is needed in its implementation, because all SRAMs are single-ported. In fact, parallel $s=3$ is 10% less hardware-consuming than parallel SEP, because large single-ported SRAMs from the shared part provide better area efficiency results as they multiplex the bitlines to share sense amplifiers.

SRAM time results show that multiset schemes are about 12% slower than the parallel SEP design, because of double-ported SRAMs. However, the parallel $s=3$ signature is about 5% slower, because of the large single-ported SRAMs. Energy values show that an increment in area for multiset signatures translates into an increment in energy consumption, but such an area increment comes from bitlines and wordlines whose energy consumption is less significant than

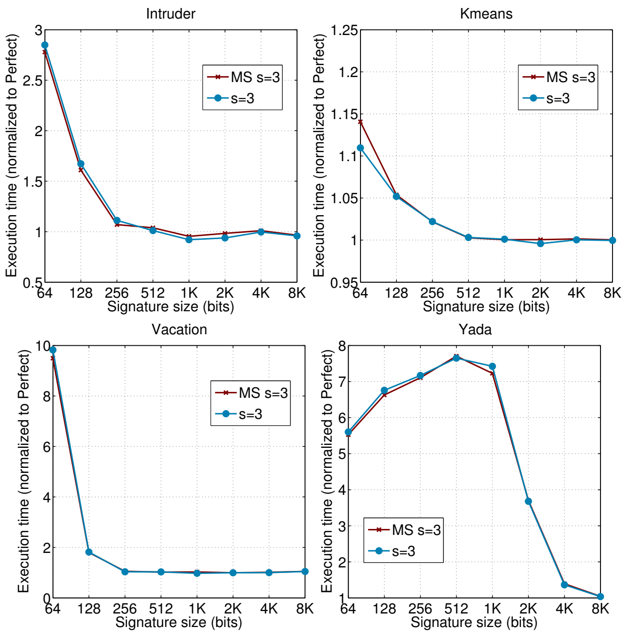


Figure 6.19: Execution time normalized to perfect signatures comparing parallel multiset $s=3$ and parallel $s=3$ signatures.

other components in the SRAMs. Thus, parallel MS $s=3$ signatures are 6% more energy-consuming than parallel SEP ones, while parallel $s=3$ scheme saves up to 19% of energy with respect to parallel SEP.

As regards hash functions, combinational logic modules were implemented in Synopsys Design Compiler using the H3 matrices of the simulation. These matrices show 13 bits per column on average (half of 26 address bits) and they have many bit pairs in common. Synopsys Design Compiler optimized the XOR gate trees finding these bit pairs to use as few gates as possible. Table 6.2 shows the results of these optimizations compared to the upper bound values estimated in Section 6.1.5, multiplied by $3.6\mu m^2$ which is the area of 2 fan-in XOR gates in Synopsys. Also, the optimized version uses 3 and 4 fan-in gates. Comparing H3 hash and SRAM area in Table 6.3, hash logic is about one twentieth of the SRAM area for parallel SEP and $s=3$ 2Kbit signatures. Note that the hash logic grows linearly with the filter size, so the fraction of H3 hash logic with respect to SRAMs is more negligible as filter size grows. Locality-sensitive hashing (not shown) uses about 12.5 bits per column of hash matrices on average, thus reducing area up

Table 6.2: Implemented synopsys optimized XOR hash area versus estimated unoptimized upper bound area using 2 fan-in XOR gates (in μm^2).

Filter 4Kb ($m = 12$)	**Implemented**	**Estimated**
Regular SEP	945.36	2073.6
Regular MS	1709.28	4492.8
Parallel SEP	822.96	1728
Parallel MS	1494.71	3801.6
Parallel MS $s=3$	994.32	2376

to 3% with respect to generic H3 implementation.

Table 6.3 shows the area required by PBX implementation of hash functions as a hardware cost lower bound for H3-type signatures. PBX is about four times as concise as H3 for parallel SEP and $s=3$ signatures, because only an XOR gate per index bit is needed instead of a whole XOR tree of height four. Table 6.3 shows a fourfold increment in time and energy for H3 implementation of hash functions compared to PBX. PBX exhibits best implementation figures and performs similarly to H3 [111], however, PBX lacks generality, because a previous study of the entropy of workloads must be carried out to enable better signature performance. Last but not least, H3 delay could be hidden by pipelining the index generation and the access to the SRAMs.

Table 6.3: Area (μm^2) figures for separate, multiset and asymmetric signatures, as well as time of the critical path (ns) and dynamic energy per access to both RS and WS (pJ). 65nm technology. $k = 4$ hash functions.

			Area (μm^2)			**Time (ns)**			**Energy (pJ)**		
Filter size (2^m)			**2Kbit**	**4Kbit**	**8Kbit**	**2Kbit**	**4Kbit**	**8Kbit**	**2Kbit**	**4Kbit**	**8Kbit**
Regular SEP	HASH	PBX	215	233	246	0.10	0.10	0.10	0.15	0.16	0.17
		H3	883	945	1021	0.47	0.44	0.46	0.78	0.84	0.90
	SRAM		105537	174097	314426	0.47	0.55	0.62	11.73	15.79	26.48
Regular MS	HASH	PBX	250	264	286	0.10	0.10	0.10	0.17	0.18	0.19
		H3	1607	1709	1828	0.47	0.48	0.48	1.36	1.47	1.57
	SRAM		299892	543369	964564	0.72	0.90	1.21	23.90	40.33	63.71
Parallel SEP	HASH	PBX	181	199	215	0.09	0.10	0.10	0.12	0.14	0.15
		H3	737	823	883	0.43	0.43	0.47	0.67	0.72	0.78
	SRAM		16133	26619	48038	0.37	0.40	0.40	2.38	3.67	5.77
ASYM	HASH	PBX	181	199	217	0.09	0.10	0.10	0.12	0.14	0.15
		H3	1143	1270	1350	0.45	0.49	0.48	0.97	1.08	1.14
	SRAM		16133	26619	48038	0.37	0.40	0.40	2.38	3.67	5.77
Parallel MS	HASH	PBX	214	230	250	0.09	0.09	0.10	0.14	0.15	0.17
		H3	1384	1495	1607	0.46	0.48	0.47	1.16	1.28	1.36
	SRAM		36771	65816	110672	0.42	0.43	0.47	4.35	8.30	11.25
Parallel MS $s = 3$	HASH	PBX	156	168	181	0.09	0.09	0.09	0.09	0.10	0.11
		H3	915	994	1078	0.41	0.48	0.47	0.78	0.84	0.91
	SRAM		19175	34468	58397	0.42	0.43	0.47	2.46	4.24	5.76
Parallel $s = 3$	HASH	PBX	151	163	176	0.09	0.09	0.09	0.09	0.10	0.11
		H3	768	816	872	0.41	0.45	0.44	0.64	0.68	0.72
	SRAM		14015	24669	42738	0.40	0.41	0.43	1.97	3.08	4.40

7 Scalability Analysis

In this chapter we perform a scalability study of signatures in the HTM system discussed in Section 3.1.3. Throughout this thesis, we have been testing our proposals with the STAMP benchmark suite and fixed parameters, while varying the signature size. However, in this chapter we keep the size of the signature constant and vary the parameters of the workload. Although the STAMP benchmarks can be parameterize to exhibit different transactional characteristics, like increased contention or different transaction lengths, such characteristics cannot be easily decoupled from each other, so we have used other Stanford benchmark specially developed for orthogonal characteristic analysis, called EigenBench.

EigenBench is proposed by Hong et al. [46] as a lightweight microbenchmark that can emulate a set of orthogonal application characteristics which are useful in understanding the performance of a TM system. It can, for example, separate the effect of working-set size from the size of transactions which are characteristics that are usually tied each other. Besides orthogonal characteristics, EigenBench is able to mimic application-based benchmarks whose memory patterns are known in advance and represent realistic workloads. Furthermore, we can generate performance pathologies [9] that can be detected in several TM systems.

Regarding the signature size, we have chosen 4Kbit filters per data set and 4 hash functions in our experiments. Sanchez et al. [92] perform a study of signatures in real systems, where they use such sizing values for the signatures. They estimate a die size increase of 0.10% in an AMD Barcelona quad-core CMP without multithreading (0.25% per core), and a die size increase of 1.1% for the Sun Niagara 8-core, 4-way multithreading CMP (4.1% of core size increase, because we need a signature per thread context). They conclude that those area numbers are relatively small (half of the instruction cache area in Niagara) compared to the overall processor core area.

The remainder of the chapter is organized as follow. Section 7.1 discusses the EigenBench implementation. Section 7.1.1 describes the modifications that we have made to EigenBench in order to include spatial locality and to adapt it to the simulation environment. Section 7.1.2 deals with the orthogonal characteristics that can be expressed with EigenBench. Section 7.2 discusses the results obtained for each characteristic we have explored: contention in Section 7.2.1, transaction length in Section 7.2.2 and concurrency in Section 7.2.3. Finally, Section 7.2.4 deals with the working set problem of EigenBench.

7.1. The EigenBench Benchmark

EigenBench is a simple algorithm to generate random memory access patterns. The pseudocode of its core is shown in Figure 7.1. There are two global arrays: a *hot* array which is shared between all threads and accessed transactionally; and a *mild* array, which is also accessed within a transaction, but each thread works on its own array partition, so accesses will not cause conflicts. Sizes of the arrays, `N_HOT` and `N_MILD`, are configurable parameters of the application, as well as the arguments of the `test_core` function.

The core transaction, lines 8–25, performs a set of read and write memory accesses to the global arrays. Specifically, `total` is the number of accesses that are executed per transaction. The `total` variable, in line 5, results of the summation of `R_HOT`, `W_HOT`, `R_MILD` and `W_MILD` application parameters, which hold the number of read and write actions to be performed on the hot and mild arrays. Function `rand_action`, lines 10 and 29, randomly chooses between reading or writing the arrays, and decrements the variable corresponding to the action chosen. Such variables are previously instantiated with the application parameters (see line 7). Then, depending on the action, the transaction reads or writes a random location of one array (lines 11–22). The `rand_index` function calculates the random location index within the limits of the chosen array. If application parameter `lct` is not zero, then an already accessed index is randomly chosen from the history buffer, a local array that holds the last accessed array location indexes, with `lct` probability. Finally, once the transaction has committed, EigenBench performs `R_OUT` + `W_OUT` operations outside the transaction before executing the next transaction. A total of `loops` transactions are executed.

```
global long array_hot[N_HOT];
global long array_mild[N_MILD];
void test_core(tid, loops, lct, R_HOT, W_HOT, R_MILD, W_MILD, R_OUT, W_OUT) {
  long val=0;
  long total = W_HOT + W_MILD + R_HOT + R_MILD;
  for (i=0; i<loops; i++) {
    (r_hot, w_hot, r_mild, w_mild) = (R_HOT, W_HOT, R_HOT, W_MILD);
    BEGIN_TM();
    for (j=0; j<total ; j++) {
      switch(rand_action(r_hot, w_hot, r_mild, w_mild)) {
        case READ_HOT:
          index = rand_index(tid, lct, array_hot);
          val += TM_READ(array_hot[index]);
        case WRITE_HOT:
          index = rand_index(tid, lct, array_hot);
          TM_WRITE(array_hot[index], val);
        case READ_MILD:
          index = rand_index(tid, lct, array_mild);
          val += TM_READ(array_mild[index]);
        case WRITE_MILD:
          index = rand_index(tid, lct, array_mild);
          TM_WRITE(array_mild[index], val);
      }
    }
    END_TM();
    val += local_ops(R_OUT, W_OUT, val, tid);
  }
}
action rand_action(r_hot, w_hot, r_mild, w_mild) {
// With uniform random probability based on r_hot, w_hot, r_mild, w_mild
// randomly choose one among: READ_HOT, WRITE_HOT, READ_MILD, WRITE_MILD.
// Then, decrease corresponding variable (r_hot, r_mild,...) by one.
}
long rand_index(tid, lct, array) {
// With probability of lct, choose a saved index from the history buffer, or
// randomly choose an index from range [0, N_HOT-1] or [tid*N_MILD,
// (tid+1)*N_MILD-1] and save it to the history buffer.
}
long local_ops(r_out, w_out, val, tid) {
// Perform r_out reads and w_out writes on a private array in random order.
}
```

Figure 7.1: Pseudocode of EigenBench core function.

7.1.1. Modifications to EigenBench

We have modified EigenBench to adapt it to the simulation environment described in Section 3.1, and to simulate spatial locality of reference.

EigenBench is released to work with TL2 [29], an STM system where transactional accesses must be explicitly annotated. The `TM_READ` and `TM_WRITE` instructions showed in Figure 7.1 are used to do so. Hence, other non-annotated instructions are not tracked by the STM system. However, we use an implicit HTM system (see Section 3.1) where all instructions enclosed by a transaction are implicitly taken as transactional. Then, calls to `random` function inside `rand_action` and `rand_index` functions are tracked by the transactional system provoking the serialization of transactions. To solve it, we used a Mersenne twister pseudorandom generator per thread, as pointed out in Section 3.2.2.

EigenBench, as is, generates random memory traces that can be biased by the `lct` parameter to introduce temporal locality of reference with a given probability. We have modified the benchmark to include spatial locality of reference. We have defined the parameter `lcs` as the probability that an access is nearby located to a preceding access. For the spatial locality distribution we have used the notion of *random walk* introduced by Thiébaut et al. [99]. The sequential accesses of the program to memory can be modeled as a random walk through a one-dimensional integer array. This integer array is main memory, the walker is EigenBench, and the jumps correspond to the gaps between consecutive accesses. The length of each jump is a sample value of the random variable X with the following probability distribution:

$$Pr[X > u] = \left(\frac{u_0}{u}\right)^{\theta},$$

where $u > 0$, and u_0 and θ are constants. The parameter θ describes the spatial locality of the random walk. As θ increases, the walk gets more locally distributed. We have chosen $\theta = u_0 = 1$, so that the random walk is governed by the simplest form of the Zipf distribution [52], where the first most common jump is of length $u = 1$, second most common jump ($u = 2$) occurs 1/2 as often as the first, the third most common jump ($u = 3$) occurs 1/3 as often as the first and so on.

Figure 7.2 shows the pseudocode of the modification to EigenBench including the locality random walk. We have limited the jumps to a length of sixteen. Thus, jumps of length 1 have a probability of 0.3, while the probability of jumps of length 2 is 0.15, 0.1 for length 3, and so forth. To get such a Zipf distribution from a random distribution that equiprobably yields numbers between 0 and 1023, we have defined an array, in lines 2–3, with the boundaries of the intervals for each jump following the probabilities above. If the random number, in line

```
1  long history_buffer[N_HB];
2  int zipf = {303, 454, 555, 631, 692, 742, 785, 823, 857, 887, 915, 940, 963,
3              985, 1005, 1024};
4  long rand_index(tid, lct, lcs, array) {
5    ... // Original code
6    if(// generate a locality random walk with probability lct) {
7      int sign = random([-1, 1]); // The jump can be positive or negative
8      int rand = random([0 1023]); // A random number between 0 and 1023
9      // If rand is in [0, 303) the jump is 1. If in [303, 454) the jump is 2, ...
10     for(jump=1; jump<=16; jump++)
11       if(rand < zipf[jump-1]) break;
12     addr = top(history_buffer); // Get the last accessed location
13     x = (addr+sign*jump); // Perform the jump
14     push(hist, x); // Insert the new accessed location in the history buffer
15     return x;
16   } }
```

Figure 7.2: Pseudocode of the function that generates the locality random walk.

8, is lower than the first interval boundary, i.e. `rand` $\in [0, 303)$, then the jump is of length 1. If `rand` $\in [303, 454)$, the jump is of length 2, and so on. The length of the jump is calculated in lines 10–11. At the end of the `for` loop, the variable `jump` holds the length of the jump to be performed, so the following lines get the last accessed location from the history buffer, and the jump is added to it, thus forming the new location to be accessed, which is inserted in the history buffer and then returned. Note that the jump can be randomly added to or subtracted from the last address accessed (lines 7 and 13).

7.1.2. Orthogonal TM Characteristics

EigenBench can be used to simulate a given execution pattern that exhibits a series of orthogonal TM characteristics. Hong et al. [46] define a set of *eigen-characteristics* that are orthogonal each other, but they can be used combined to express more conventional non-orthogonal characteristics. The eigen-characteristics are the following:

- *Concurrency*: It defines the number of concurrently running threads of the application.
- *Working-set Size*: It is the size of the used memory. Can be expressed as the summation of the size of the arrays used by EigenBench.
- *Transaction Length*: Defined as the number of reads and writes inside a

transaction, it can be worked out by adding `R_HOT`, `W_HOT`, `R_MILD` and `W_MILD`.

- *Pollution*: The number of writes (`W_HOT`+`W_MILD`) with respect to transaction length is defined as pollution.

- *Temporal Locality*: It is the probability of repeated addresses per shared accesses.

- *Contention*: The probability of conflict of a transaction. See Section 7.2.1.

- *Predominance*: It is defined as the fraction of transactional access cycles to total execution cycles.

- *Density*: It measures the fraction of non-shared cycles executed outside transactions to total non-shared cycles, out and inside transactions.

In next section, we study the behavior of our proposals with EigenBench varying contention, transaction length and concurrency characteristics. Spatial locality is added to the eigen-characteristics above, and its effect is also discussed.

7.2. Experimental Evaluation

We used the simulated target system described in Section 3.1.3 as the base HTM system for the experiments. Signature size was set to 4Kb per data set and 4 hash functions for imperfect signatures. All experiments show speedup of our signature proposals with respect to the unprotected (neither locks nor transactions) serial version. Also, the unprotected concurrent version is shown, which is an upper bound of available performance. Of course, such a bound is not achievable in the presence of contention, since the protection of locks or transactions would serialize the conflicting critical section. Finally, perfect signatures and parallel signatures are shown for comparison.

Section 7.2.1 studies the effect of contention, Section 7.2.2 discusses transaction length, and Section 7.2.3 deals with concurrency in transactional memory.

7.2.1. Contention Results

Contention is defined in [46] as the probability of conflict of a transaction, and an approximate expected value is proposed:

$$P_{conf} = 1 - \left(1 - min\left\{1, \frac{(N_{TH}-1)W'_{HOT}}{N_{HOT}}\right\}\right)^{W'_{HOT}+R'_{HOT}}. \qquad (7.1)$$

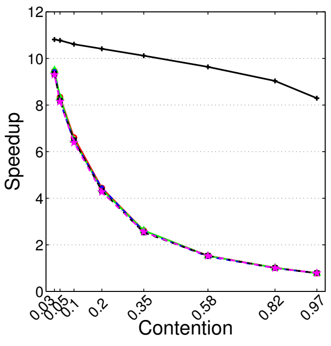
Expression 7.1 is deduced as follows. Let W'_{HOT} and R'_{HOT} be the number of accesses to different addresses in the hot array. R'_{HOT} can be defined as

$$R'_{HOT} = \begin{cases} 1 & \text{if } lct = 1 \\ \lceil (1-lct)R_{HOT} \rceil & \text{otherwise} \end{cases},$$

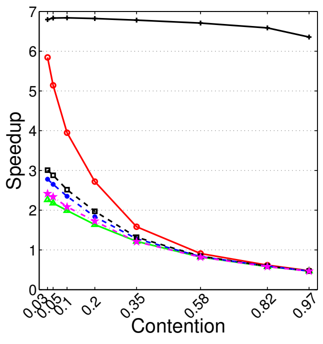
and W'_{HOT} is defined in the same way. If we have N_{TH} threads and the hot array length is N_{HOT}, then the probability that an access in a transaction causes a conflict is $(N_{TH}-1)W'_{HOT}/N_{HOT}$, which stands for the number of writes performed by the other transactions divided by size of the array. It is supposed that $N_{HOT} \gg W'_{HOT}$. Then, $1 - ((N_{TH}-1)W'_{HOT}/N_{HOT})$ is the probability that an access does not cause a conflict, which happens $W'_{HOT} + R'_{HOT}$ times. The complement of that is the probability of a conflict of Expression 7.1.

Table 7.1 shows the parameters of EigenBench for the analysis of contention. Contention ranges from 0.03 to 0.97 by varying the size of the hot array, N_{HOT}, from 1K to 128K `long` elements. We test two configurations. One with short transactions and another with long transactions, which in turn is tested with different values of spatial locality, $lcs \in \{0, 0.25, 0.5, 0.75\}$. Predominance is kept at 80% with the given R_{OUT} and W_{OUT} values. Each thread of the parallel version executes $loops = 128$ transactions, while the serial version executes $128 * 15$ transactions.

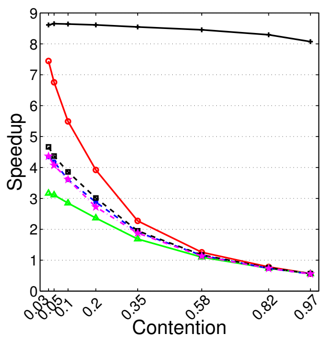
Figure 7.3 shows the results obtained from the simulator. Figure 7.3a depicts the results of the parameter configuration that defines short transactions. We can see that the unprotected version of the code do not achieve the maximum speedup available, which is 15. Instead, it is 11× as fast as the serial version. The problem lies in the implementation of EigenBench, as the mild array is of size $N_{MILD} * N_{TH}$, so the serial version works with a mild array of size N_{MILD}, whereas in the concurrent version, the mild array is 15 times larger. Then, the cache hierarchy makes the serial version goes faster than the parallel one (see Section 7.2.4 for a further explanation). Also, the network traffic increases since the hot array is shared between 15 cores, and this gets worse as contention is higher. Figure 7.3a also shows that all signature variants perform the same as perfect signatures when transactions are short and signature size is large enough



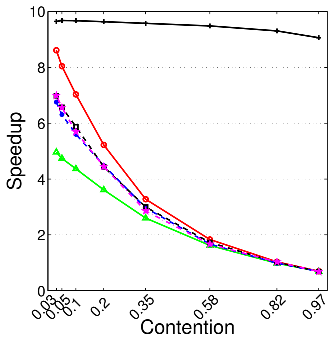
(a) Xact length: Short



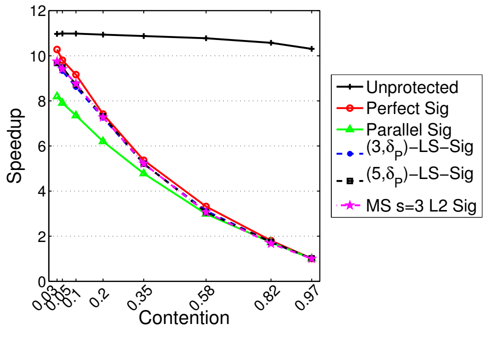
(b) Xact length: Long. LCS: 0%



(c) Xact length: Long. LCS: 25%



(d) Xact length: Long. LCS: 50%



(e) Xact length: Long. LCS: 75%

Figure 7.3: Contention results for 15 threads.

(4Kbit in this case). Notice that our proposals do not harm the performance of short transactions.

Next experiments use long transactions, and signature length is kept at 4Kbit per data set. Table 7.1 shows the parameters we used. R_{MILD} and W_{MILD} changed to 200. The rest of parameters keep the same except for R_{OUT} and W_{OUT} that were modified to maintain 80% of predominance. Figure 7.3b shows the results obtained without spatial locality. The speedup now drops significantly due to the aforementioned issues. However, our proposals perform better than the parallel signature version since, although the accesses are randomly distributed, some of them are arbitrarily nearby enough to take advantage of locality-sensitive signatures. Figures 7.3c, 7.3d and 7.3e show results in which spatial locality is set to 25%, 50% and 75% respectively. The speedup of all versions improves as locality increases, since the cache hierarchy is better harnessed. Also, the system works at 64 byte block granularity, so certain accesses nearby each other will be in the same cache memory block as the arrays comprise *long* elements of 4 bytes each. Thereby, spatial locality implies some amount of temporal locality which improves the performance. In any case, our signature proposals perform similar to or better than the parallel signature in the explored cases, and they practically match the performance of perfect signatures when locality is 75% in Figure 7.3e. Finally, note that too much contention can lead the HTM system to perform worse than the serial version.

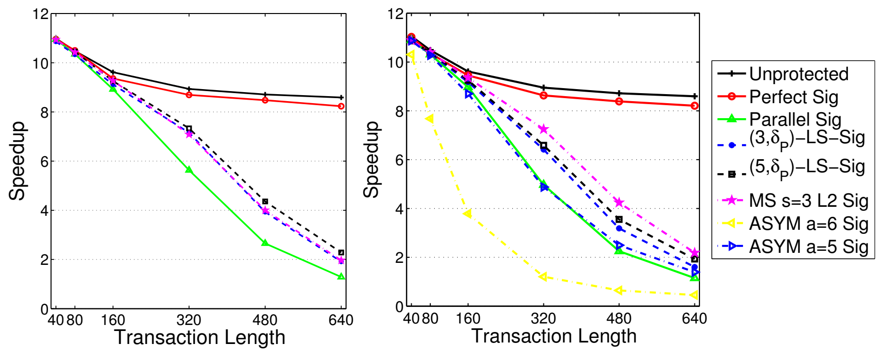
7.2.2. Transaction Length Results

To study the effect of transaction length we used the parameters showed in Table 7.1. Again, we have two configurations. In the first one, which we have called *symmetric*, transactions read the same number of locations than they write, while in the second one, *asymmetric*, there are three times more reads than writes. We have set R_{HOT} and W_{HOT} to zero in order to have no contention. Also, predominance is 100% as $R_{OUT} = W_{OUT} = 0$. Locality has been set to 25%.

Figure 7.4a shows the results for the symmetric configuration parameters in Table 7.1. Maximum speedup is about $11\times$ the serial for the shortest transaction length of 40 elements. As we discussed in the last section, this is due to the working set effect discussed in Section 7.2.4. Perfect signatures perform similar to the unprotected version in this case, as contention is set to 0. However, a small performance drop can be appreciated as transaction length increases, due to a small fraction of aborts that might be caused by false sharing. The

Table 7.1: Parameters of EigenBench for the experiments.

	Contention		**Xact Length**		**Concurrency**	
Param	Xact Length: Short	Xact Length: Long	Sym	Asym	Non-local	Local
N_{HOT}	15	15	15	15	[1, 15]	[1, 15]
loops	128	128	128	128	[1920, 128]	[1920, 128]
N_{HOT}	$[1K, 128K]$	$[1K, 128K]$	128K	128K	128K	128K
N_{MILD}	1M	1M	1M	1M	1M	1M
R_{HOT}	45	45	0	0	45	45
W_{HOT}	5	5	0	0	5	5
R_{MILD}	45	200	$[20, 320]$	$[30, 480]$	200	200
W_{MILD}	5	200	$[20, 320]$	$[10, 160]$	200	200
R_{OUT}	18	49	0	0	49	49
W_{OUT}	2	41	0	0	41	41
lct	0	0	0	0	0	0
lcs	0	0, 0.25, 0.5, 0.75	0.25	0.25	0	0.5



(a) Symmetric (b) Asymmetric

Figure 7.4: Transaction length results for 15 threads.

results for imperfect signatures, the parallel one and our proposals, get affected by false conflicts due to false positives in the filters. The performance with parallel signatures drops quickly from transaction length 160 onwards. Parallel signatures of 4Kbit per set match the performance of the serial version for transaction length 640. However, our signature schemes perform better than parallel signatures, although they exhibit a considerable performance degradation with respect to perfect signatures from transaction length 480 onwards.

Figure 7.4b depicts the results obtained for different transaction lengths and asymmetric data sets. Now, the read set is three times as large as the write set,

Table 7.2: Average read set and write set lengths ($|RS|$ and $|WS|$) measured by the HTM system compared to that of EigenBench input parameters. Transaction length is $|RS| + |WS|$, and the RS to WS ratio is $\frac{|RS|}{|WS|}$.

Parameters				**Measured**			
$\lvert RS\rvert + \lvert WS\rvert$	$\lvert RS\rvert$	$\lvert WS\rvert$	$\frac{\lvert RS\rvert}{\lvert WS\rvert}$	$\lvert RS\rvert + \lvert WS\rvert$	$\lvert RS\rvert$	$\lvert WS\rvert$	$\frac{\lvert RS\rvert}{\lvert WS\rvert}$
40	30	10	3	79.2	55.2	24.0	2.3
80	60	20	3	138.6	94.8	43.8	2.2
160	120	40	3	243.2	161.3	81.9	2.0
320	240	80	3	391.2	265.0	126.2	2.1
480	360	120	3	531.5	367.4	164.1	2.2
640	480	160	3	671.6	469.5	202.0	2.3

as seen in Table 7.1. We can see that the best results using imperfect signatures are yielded by our MS $s = 3$ L2 signatures, that can cope with the data set asymmetry. Reconfigurable asymmetric signatures have been also tested, so that the configuration parameter a is 5 and 6. With a RS to WS ratio of 3, ASYM $a = 6$ should achieve the best results. However, performance is very poor for such a configuration, and ASYM $a = 5$ gets better results. This is because of the HTM system, which is an implicit HTM system where every memory access enclosed by a transaction is implicitly tracked by the TM system. Reads and writes to the hot and mild arrays, which are the accesses that we used to get the transaction length, are not the only memory accesses within transactions since `rand_index`, `rand_action`, and other control code perform memory accesses that are tracked by the TM system. Therefore, we show in Table 7.2 the real RS and WS lengths measured by the HTM simulator, and the corresponding transaction length taken from the input parameters of EigenBench. Note that the transaction length is longer when using an implicit HTM system. Now, the RS to WS ratio is not three as inferred by the input parameters. Instead, the ratio is about two, which is closer to $\frac{5}{3}$, the ratio of ASYM $a = 5$ filters, than to $\frac{6}{2}$, which is the ratio of ASYM $a = 6$ filters. Thus, ASYM $a = 5$ yields better results for reconfigurable asymmetric signatures.

7.2.3. Concurrency Results

In this section we study the scalability of signatures in terms of concurrency. The parameters in last columns of Table 7.1 were used to perform the experiments. Figure 7.5a shows the results obtained in the absence of spatial locality. We get poor results because of the working set effect explained in Section 7.2.4. We have measured and increasing speedup of our $(5, \delta_P)$-LS signature with re-

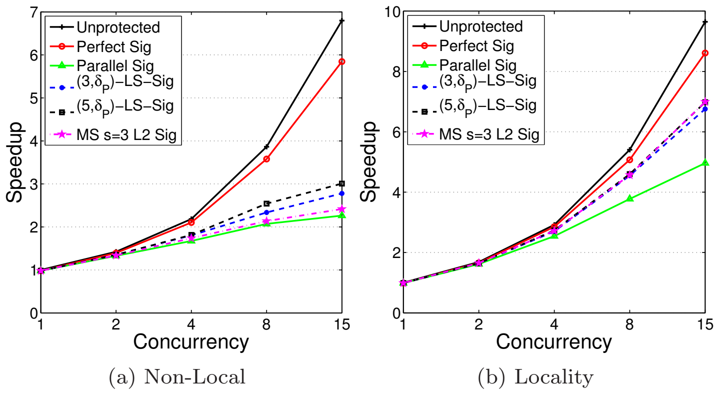


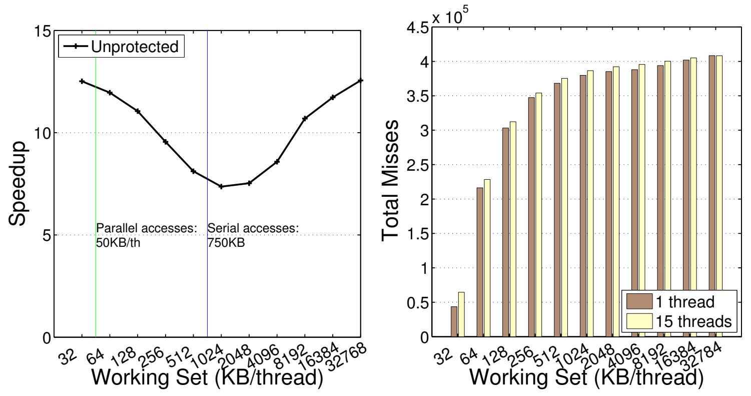
Figure 7.5: Concurrency results for 15 threads.

spect to the parallel conventional signature of $[1, 1.02, 1.08, 1.23, 1.33]$, for 1, 2, 4, 8 and 15 threads of concurrency respectively.

The results depicted in Figure 7.5b include the locality parameter set to 50%. Now, we obtain better results when comparing our $(5, \delta_P)$-LS signatures to parallel conventional signatures. In this case, we get a relative speedup of $[1, 1.02, 1.08, 1.23, 1.41]$. We can see that our signature proposals scale better than the parallel conventional signatures. Also, it seems that the more cores are available, the worse the effect of false conflicts in imperfect signatures is. So, better signatures will be needed for future many-core processors.

7.2.4. The Working Set Effect

Figure 7.6 shows speedup and total misses (L2 misses) of the unprotected version of EigenBench varying the working set characteristic. The parameters used in these experiments are all set to 0 except N_{MILD} which varies from 8K elements to 8M elements per thread. The working set showed in Figures 7.6a and 7.6b are worked out by multiplying the values of N_{MILD} by 4, since each element of the mild array is a *long* value of 4 bytes. Also, R_{MILD} and W_{MILD} are set to 50 each. Then, as parameter *loops* is set to 1920 for the serial version, $1920 * 50 * 2 * 4bytes = 750KB$ are read or written by the serial version of EigenBench (the blue line in Figure 7.6a). In the parallel version, each of the fifteen threads performs 128 *loops*, so $128 * 50 * 2 * 4bytes = 50KB$ are read or written by each thread in the parallel version of EigenBench (the green line in



(a) Speedup (b) Total misses

Figure 7.6: Working-set results for 15 threads. N_{MILD} varies from 8K to 8M elements of 4 bytes. $R_{MILD} = W_{MILD} = 50$. The rest of parameters are set to 0.

Figure 7.6a).

Therefore, if we have a working set of 32KB, by the pigeonhole principle, the serial version will traverse its entire mild array (32KB), provided that the random generator is good, and then, it will repeat accesses to the same memory locations which are already in the cache hierarchy (L2 is 8MB, L1D is 32KB), until 750KB accesses are performed. On the other hand, each thread of the parallel version will traverse its entire mild array partition (32KB), until 50KB accesses are performed, so there will be repetitions as well. However, with a working set of 64KB, the serial version still complies with the pigeonhole principle, whereas in the parallel version, 50KB is less than 64KB and repetitions are less likely. As we increase the working set, repeated accesses to the same locations are less probable in the parallel version, while the pigeonhole principle still applies for the serial one, which explains the speedup degradation from 32KB to 1024KB. From 1024KB onwards, the 750KB accessed by the serial version can be performed without repetition, and the serial version begins resembling the parallel version until both are accessing memory locations without repetitions. Note that the speedup curve decreases and increases gradually, but not abruptly. One might expect a heaviside step function once the working set surpasses the blue line that represents the number of memory bytes accessed by the serial version. However, the imperfect random generator and the effect of cache block granularity make that repeated accesses still persist and disappear gradually.

In Figure 7.6b, we can see how L2 cache misses increase with the working set. Having a block size of 64 bytes and array elements of 4 bytes, an access to an element of the array brings to cache 15 contiguous elements of the array that will not cause cache misses. As the working set increases, those 15 elements of the block are less likely to be accessed because of the random access pattern and the non-fulfilment of the pigeonhole principle. Also, the parallel version accesses more different memory locations than the serial one when the working set is small, and therefore, more compulsory misses are accounted. When the working set is 32MB the number of misses equalizes.

Conclusions and Future Work

In the era of multicore processors, there is an increasing interest in finding new programming paradigms that can ease the writing of concurrent programs. Harnessing the power available in these machines has turned out to be a complex task, specially when programmers are used to dealing with uniprocessor systems. In order to tackle this problem, Transactional Memory emerges as an alternative to conventional multithreaded programming to ease the task of exploiting multicore processors. As a matter of fact, Hardware Transactional Memory is gaining popularity to such an extent that main processor manufactures like Intel, AMD and Sun Microsystems are working to include transactional extensions to their new chip multiprocessor models.

In the context of Hardware Transactional Memory, we have proposed several optimizations of the conflict detection mechanism, which has been proved to be a critical part of the system. Specifically, our proposals focus on novel signature implementations that are used in helping the virtualization of the transactional system.

The first signature scheme proposed in this thesis is the *Interval Filter*. It is an alternative to Bloom-based signatures that holds the addresses of the memory locations accessed by a transaction in terms of intervals. The Interval Filter manages to outperform the conventional Bloom signature scheme when the applications exhibit a locality access pattern such that a few large intervals of contiguous addresses can be created. This is the case of certain benchmarks of the STAMP benchmark suite like Kmeans and Labyrinth. However, our signature alternative fails to perform better than Bloom signatures in the presence of lots of random single accesses or small intervals, since it is implemented as a fully associative memory that cannot expand too much if we want it to keep as concise as possible. Consequently, the Interval Filter is not a general solution but could be an option in case of embedded systems where we are able to know the access pattern of the application in advance.

The next contribution of this thesis is in the line of harnessing spatial locality, and it is based on Bloom filter signatures, which provide us with a more general solution than that achieved by Interval Filters. We have named it *Locality-Sensitive Signatures*, which define new maps for the hash functions to reduce the number of bits inserted in the filter, for the addresses with spatial locality. That is, nearby memory locations share some bits of the Bloom filter. As a result, false conflicts are significantly reduced in transactions that exhibit spatial locality in their read or write sets, but the false conflict rate remains unalterable for transactions that do not exhibit locality at all. We propose several Locality-Sensitive Signature variants and we can conclude that most of them perform equal or better than conventional signatures. However, we find that the δ_P version of Locality-Sensitive Signatures, which is define piecewise with different granularity per hash function, beats the rest of signatures in that it behaves more evenly whatever the benchmark, and outperforms most of them in most of cases. Finally, as Locality-Sensitive Signatures are based on new locality-aware hash maps, their implementation does not require extra hardware, moreover, we can save up to 3% of XOR logic gates.

Last proposals we contribute with tackle the asymmetry in transactional data sets. Conventional signatures track read and write accesses with two separate, same-sized Bloom filters, whereas transactions frequently exhibit read and write sets of uneven cardinality, which introduces inefficiencies in the use of signatures. The first optimization we propose in this context is the *Multiset Signature*. It uses a single Bloom filter to track both read and write sets, and thereby the false positive rate of both sets tend to be similar each other. We find that Multiset Signatures outperform conventional signatures in most cases, except for transactions whose size exceeds a given threshold. Also, as Multiset Signatures need double-ported SRAMs for their implementation, we propose an improvement in order to reduced the amount of hardware needed by multiported SRAMs, whose size grows quadratically with the size of the array. Given that a considerable amount of data is both read and written within a transaction, we proposed *Multiset Shared Signatures*, where certain filters of the parallel implementation of the Multiset Signature have only one hash function shared between the read set and the write set, while the rest maintains two hash functions that map addresses of read and write sets into the same filter. This scheme has a part of the signature that is unable to distinguish between read and write accesses, which introduces a source of read-read dependencies, and a part that is able to differentiate them. Multiset Shared Signatures are improved by including the aforementioned Locality-Sensitive feature averaging a 47% increase in performance without increasing the hardware needed for their implementation.

A different approach to deal with asymmetry in transactional data sets is our *Reconfigurable Asymmetric Signature*. It starts from a conventional parallel signature where the various subarrays that comprise the signature can be configured to belong to the read set or to the write set. Thus, via a reconfiguration mask register, we can configure the signature to have the read filter larger than, equal to or smaller than the write filter. Such a configuration can be performed on either a per-transaction or a per-application basis. We propose a heuristic to configure the signature on a per-application basis that involves profiling the application to get the average read set to write set ratio. We found that such a heuristic is a good although non-optimal approach to get the most out of our Reconfigurable Asymmetric Signatures.

Finally, we study the response of the conventional signature proposals compared to our signature optimizations when we stress the transactional system in terms of contention, transaction length, and number of concurrent transactions. We have used a novel benchmark for the orthogonal characteristic study of TM systems, so-called EigenBench, which has been modified to include the modelling of spatial locality of reference. The results show that contention can seriously harm the execution of transactional code to such an extent that the parallel version of the workload performs worse than the serial version. Our signature proposals can ameliorate the performance degradation of the system, most importantly when the workload exhibits a certain amount of spatial locality. Unfortunately, both imperfect signatures and perfect signatures converge, thus limiting the room for improvement in imperfect signature development. As regards transaction length, we also obtain better results than conventional signatures designs in most cases. However, we can note a trend towards achieving the same execution time than conventional parallel signatures as transaction length grows. In this case, false positives can noticeably harm the execution compared to the perfect signature. Results also suggest that implicit transactional memory systems, although easier to program, might have unpredictable effects in transactional length that can mislead the configuration of reconfigurable asymmetric signatures, if it is carried out on theoretical parameters instead of profiling. Also, it seems that the more cores are available, the worse is the effect of false conflicts in imperfect signatures. Our signature optimizations scale better than parallel conventional signatures.

Transactional Memory has proved to be a novel paradigm which is being adopted by main processor manufactures. They are introducing in their chips a best effort approach that executes transactions as long as there are resources to support them. In this thesis, we have proposed optimizations to signatures that are used to overcome the limitations of best effort transactional systems.

However, there is still work to be done in relation to such a virtualization of the transactional system. To conclude this thesis we can point out the following future work paths that we think are worth exploring:

- Our proposal of reconfigurable asymmetric signatures were tested by using a per-application configuration strategy which turned out to be a good solution for the problem of asymmetry in transactional data sets. However, it would be worth studying a per-transaction basis reconfiguration of the asymmetric signatures, since the read set to write set ratio can change between transactions within the same application. On the one hand, we could use profiling to reconfigure the signature on the beginning of each transaction. This approach need the definition of a new instruction of the ISA to configure the signature, and might depends on the input parameters of the application. On the other hand, we could let the transactional system reconfigure the signature based on information gathered during the execution, like history of former transactions or statistics of the signature on abort.

- Some results obtained in this thesis expose a problem in transactional memory that must be addressed if we want it to succeed as a convenient alternative to conventional parallel programming. In case of high contention, transactional memory can perform worse than the serial version of the application. Research has been done to reveal the pathologies that can affect hardware transactional memory [9], and this research line is ongoing with several works that are aware of the contention problem [73, 102]. In the context of signature conflict detection transactional memory, we think it is worth studying how the critical execution of signatures because of high false conflict rates could be detected. Once detected, subsequent serialization of transactions whose signatures got spoiled by too much false positives could be a solution.

- As we have seen in the last chapter of this thesis, implicit transactional systems include redundant memory locations, like private variables, that pollutes the signatures. Yen et al. [111] and Sanyal et al. [93] tackle the problem by providing the programmer with special memory allocation constructs that define whether a location is private or shared. However, the programmer must be extremely careful as a mistake in marking objects can lead to a breakage of program correctness. Explicitly annotated transactional systems can make programming transactional memory difficult [88], which contradicts the prime purpose it was devised for. The automatic detection of private memory locations to keep them from pollute signatures in transactional memory might be a good research line to improve the performance of the system while maintaining the programmability.

- Eager-eager transactional memory systems whose conflict detection mechanism is based on Bloom signatures cannot rely on such signatures to decide whether written memory blocks must be stored into the undo log or not, since a false positive might cause not to log a block that should be logged. These systems need a log filter that holds recently logged blocks to prevent them from be stored in the log more than once within a transaction. Indeed, it is an optimizing structure as the system can operate without it. However, it could be interesting to study the effect in performance of the log filter, and if it could be possible to add a signature help to such a buffer.

Transactional Memory is a promising paradigm for simplifying the development of parallel applications. It is not clear what the future holds, but hardware industry is betting on it. This thesis is a little contribution to Transactional Memory, with the hope that it will help on clarifying its future.

Apéndice A
Resumen en español

Hace casi 50 años, Gordon E. Moore predijo que el número de transistores que podrían integrarse en un chip se doblaría cada dos años [68]. La industria del hardware ha sido capaz de cumplir la profecía de Moore durante todos estos años, desarrollando procesadores cuyo rendimiento se ha ido doblando junto con el incremento en el número de transistores. Sin embargo, en la última década, dicho rendimiento ha sido más difícil de extraer. Incrementar la frecuencia del reloj del procesador ya no es posible debido a problemas de disipación de potencia y refrigeración [74]. Por otro lado, las técnicas de extracción de paralelismo a nivel de instrucción (ILP) que se han utilizado para crear los procesadores superescalares se ven limitadas por la cantidad de paralelismo de las aplicaciones secuenciales, que raramente excede la media docena de instrucciones [107]. Como consecuencia, para suplir la falta de ILP y llenar el cauce de los procesadores superescalares, se añade la capacidad de aprovechar el paralelismo a nivel de hilo de las aplicaciones (TLP), convirtiendo estos procesadores en procesadores de múltiple hilo simultáneo (SMT) [56]. Con la aparición de los procesadores SMT, se podía intuir el cambio en la industria de los ordenadores, de la extracción automática a la extracción manual de paralelismo por parte de los programadores. Finalmente, los fabricantes de procesadores han decidido desarrollar multiprocesadores en un chip (CMPs) [35, 61] con núcleos más sencillos en lugar de grandes procesadores superescalares con SMT, y dejar en manos de los programadores la extracción del rendimiento de sus máquinas.

Con la comercialización a gran escala de los CMPs, la industria del hardware ha pasado a la comunidad del software el problema de la extracción de rendimiento de sus procesadores. Sin embargo, la mayoría de los programadores están acostumbrados a la programación serie, y la programación paralela puede llegar a resultarles compleja. El paralelismo de tareas introduce indeterminismo en el orden en que se acceden las variables de memoria, accesos que tienen que ser sincronizados en el caso de datos compartidos para evitar inconsistencias debido a condiciones de carrera entre hilos. La manera más común de sincronizar el acceso a un conjunto de datos compartidos, o lo que es lo mismo, de proteger una sección crítica del programa paralelo, es usar cerrojos. Los cerrojos serializan el acceso a las secciones críticas del programa, por lo que introducen una fuente de pérdida de rendimiento en las aplicaciones paralelas, aunque aseguran su correcta ejecución. Además, se ha de llegar a una relación de compromiso cuando se usan cerrojos. Usar pocos cerrojos sobre zonas amplias de memoria incrementa la contienda entre los hilos por adquirir el acceso a la zona crítica y aumenta la serialización, pero disminuye la sobrecarga inherente a los cerrojos y facilita la programación. Por el contrario, usar multitud de cerrojos para proteger pequeñas zonas de memoria, como simples variables, disminuye la serialización y la contienda, pero aumenta la sobrecarga y la dificultad en la programación, ya que pueden aparecer con más facilidad ciertos problemas asociados a los cerrojos, como *deadlock* o *convoying*. Todos estos problemas, unidos a la imposibilidad de composición y abstracción que ofrecen los cerrojos, hacen de la programación paralela un paradigma complejo.

La memoria transaccional (TM) [40, 45, 54] nace para facilitar la tarea de escribir aplicaciones concurrentes para CMPs, a causa de la complejidad de la programación paralela. TM toma el concepto de transacción del campo de las bases de datos y lo redefine como un bloque de instrucciones que parecen ejecutarse atómicamente y en aislamiento, permitiendo su abstracción y composición al separar la semántica de la implementación. El sistema transaccional ejecuta las transacciones en paralelo a no ser que detecte algún conflicto entre ellas, momento en el cual se serializa la ejecución. De esta manera, TM se visualiza como un modelo optimista de concurrencia, a diferencia de los cerrojos, cuyo modelo se dice pesimista puesto que la ejecución queda serializada tanto en presencia de conflictos como en su ausencia. El modelo transaccional suprime en gran medida las inconveniencias de la programación con cerrojos.

Los sistemas TM pueden implementarse en software (STM) [29, 39, 41, 44, 95] o en hardware (HTM) [3, 37, 45, 69, 84], así como puede darse una hibridación de ambas implementaciones [26, 53, 90]. Los sistemas STM muestran una notable degradación del rendimiento debido a la sobrecarga que supone implementar los mecanismos de detección de conflictos y administración de versiones en software. Ciertos aspectos de dichos mecanismos pueden ser acelerados por hardware, o por el contrario, se pueden implementar en su totalidad a nivel de chip, conformando un sistema HTM sin la sobrecarga propia del software. El mecanismo de detección de conflictos necesita almacenar todas las direcciones de las posiciones de memoria accedidas por cada transacción para detectar conflictos entre ellas, y de esa manera asegurar la atomicidad de las mismas. Mientras que el mecanismo de administración de versiones necesita almacenar las versiones nuevas o antiguas de las posiciones de memoria modificadas por cada transacción para asegurar el aislamiento.

La implementación del mecanismo de detección de conflictos en un sistema transaccional es un aspecto clave para el rendimiento del mismo. Las primeras propuestas de sistemas HTM incluyen bits de lectura y escritura transaccional en cada bloque de la jerarquía de caché, y aprovechan el hecho de que los conflictos se pueden detectar a través del mecanismo de coherencia [45, 69]. El problema de estos sistemas es que las transacciones no pueden sobrevivir a fallos de capacidad de la caché, cambios de contexto en el procesador, migraciones de hilos en el CMP y otros eventos propios de la virtualización del sistema, como fallos de página. Para poder soportar transacciones de duración y tamaño ilimitados se han propuesto las firmas de transacción para el mecanismo de detección de conflictos [14]. Estas firmas son capaces de mantener un número ilimitado de direcciones de memoria a costa de obtener falsos positivos, es decir, detectar conflictos que no existen, por lo que facilitan la virtualización del sistema transaccional. También liberan a la caché de tener que almacenar el estado transaccional de los bloques, lo que mantiene intacta una estructura tan crítica para el rendimiento del procesador como es la caché, a la vez que se facilita el cambio de contexto al tener la información transaccional localizada en la firma. El uso de estas firmas se ha extendido a multitud de propuestas de sistemas TM, tanto software como hardware [59, 65, 67, 97, 110].

El objeto de esta tesis doctoral es la optimización de las firmas de detección de conflictos en sistemas de memoria transaccional hardware. Concretamente, se proponen optimizaciones para el manejo de transacciones grandes, las cuales hacen que los falsos positivos en las firmas degraden sustancialmente el rendimiento del sistema debido a la detección de falsos conflictos. Las principales contribuciones de esta tesis en dicho campo de la memoria transaccional son las siguientes: un filtro de intervalos como alternativa a las firmas tradicionales que se implementan como filtros de Bloom; una firma sensible a la localidad, que aprovecha el principio de localidad de referencia que se muestra con frecuencia en los patrones de acceso a memoria de todo tipo de aplicaciones, para almacenar con más eficiencia las direcciones de memoria accedidas por las transacciones; una firma multiconjunto que soluciona en gran medida el problema de asimetría que presentan los conjuntos de datos accedidos por las transacciones, en los que el conjunto de datos leídos suele ser mayor que el conjunto de datos escritos, lo que lleva a desaprovechar los recursos de las firmas, que a menudo se disponen como filtros del mismo tamaño; una firma asimétrica reconfigurable, que trata el problema de la asimetría de los conjuntos de datos de las transacciones desde otro punto de vista; un estudio de escalabilidad de las firmas para la detección de conflictos con respecto a la contención, el tamaño de los conjuntos de datos y la concurrencia de transacciones en un sistema HTM; y un estudio completo con simulación y análisis de las firmas propuestas en esta tesis, utilizando herramientas y grupos de programas de prueba que son ampliamente usados y reconocidos en el ámbito de la memoria transaccional.

Las contribuciones anteriormente mencionadas han sido publicadas en conferencias internacionales [79, 81, 82, 83], workshops [80] y revistas catalogadas en el ISI Journal Citation Reports (JCR) [77, 78], y se resumen en las siguientes secciones de este anexo.

A.1. Filtro de intervalos

El filtro de intervalos (IF) [81] se propone como una alternativa a los filtros de Bloom que es capaz de reducir el número de falsos positivos en presencia de localidad. Se definen los intervalos como un conjunto de direcciones consecutivas que se pueden extraer de una traza de referencias de memoria. La Figura 4.1 muestra el diseño del filtro, que comprende n intervalos formados por un par de direcciones de memoria, donde una representa el límite inferior del intervalo y la otra representa el límite superior. Cada intervalo posee un bit de validez, $V_0, ..., V_{n-1}$, y cada límite del intervalo tiene dos líneas que ofrecen información de comparación. En el límite inferior, dichas líneas indican si la dirección a filtrar más uno es igual, $=^l_0, ..., =^l_{n-1}$, o mayor, $>^l_0, ..., >^l_{n-1}$, al valor almacenado en el límite. En el límite superior, las líneas indican si la dirección a filtrar decrementada en uno es igual, $=^u_0, ..., =^u_{n-1}$, o menor, $<^u_0, ..., <^u_{n-1}$, al valor almacenado en el límite. De esta manera, el filtro puede verse como una caché asociativa extendida.

Con la funcionalidad descrita en el párrafo anterior, el IF ofrece las mismas primitivas que el filtro de Bloom: el chequeo de pertenencia al conjunto de datos que representa el filtro, y la inserción. Para el chequeo de pertenencia se utilizan las líneas de comparación de los límites de los intervalos como indica la Figura 4.1. Sólo hay que comprobar que la línea Match esté activa para saber que hay un positivo. Así, los chequeos son rápidos, sin embargo las inserciones son más complejas como en el filtro Cuckoo-Bloom [92]. De hecho, aparecen tres casos diferenciados en la inserción de nuevas direcciones en el IF. En el primer caso, todas las líneas $=$ están a cero, por lo que la nueva dirección no limita con ningún intervalo existente y hay que crear un intervalo nuevo. Si el filtro no está Full no hay problema. Sin embargo, si no hay intervalos libres hay que agrandar uno existente, introduciendo así falsos positivos. Se busca de manera iterativa el intervalo con el límite más cercano a la dirección, y se cambia dicho límite por la dirección a insertar. En el segundo caso, una sola línea de = está activa, por lo que el intervalo correspondiente se amplia insertando la dirección original en el límite acertado. En el tercer y último caso, un $=_i^l$ y un $=_j^u$ están activos, lo que indica que la dirección a insertar es la que falta para unir dos intervalos existentes. Como resultado se funden los dos intervalos en uno y se libera uno de los intervalos utilizando su bit de validez. El algoritmo de inserción en el IF se puede ver en la Figura 4.2.

Para evaluar el filtro de intervalos se estimó que un filtro de Bloom con 4 funciones hash y un tamaño de 2K-bit requería un hardware similar a un IF con 10 entradas, para lo que se utilizó CACTI [100] y Synopsys. Para las simulaciones se utilizó el módulo Ruby de GEMS [63] para el sistema HTM, y Simics [60] como simulador de sistema completo dirigido por ejecución. Se utilizaron cuatro benchmarks de la suite STAMP [66] que muestran transacciones de gran tamaño. La Tabla 4.1 muestra los parámetros de entrada y las características transaccionales para dichos benchmarks.

La motivación para el IF viene de la Figura 4.5, en la que se muestra una clasificación por tamaño de los intervalos hallados en las transacciones de los benchmarks. En todas las gráficas se puede observar cierta cantidad de direcciones sueltas, pero la mayoría de direcciones accedidas se pueden agrupar en intervalos de tamaño mayor que uno mostrando localidad espacial. De hecho, el número de direcciones sueltas en los benchmarks está entre el 2 % de Kmeans al 22 % de Yada. En cuanto al tiempo de ejecución, la Figura 4.7 muestra los tiempos para cada benchmark, normalizados al tiempo del filtro perfecto. Se pueden observar dos modos de comportamiento en los benchmarks simulados. El IF se comporta peor que el filtro de Bloom con Bayes y Yada. Ambos benchmarks muestran el mayor número de direcciones sueltas con un 13 % y un 22 % respectivamente. Además, el intervalo más largo en Bayes es de 100 direcciones, mientras que la transacción más larga tiene 2171 direcciones. En Yada el intervalo más largo es de 11 y la transacción más larga de 578. Todo esto hace que un IF con $n = 10$ entradas se llene con facilidad y que el primer caso de las inserciones (véase la Figura 4.2) sea el más frecuente, introduciendo gran cantidad de falsos positivos que dañan la ejecución. Por otro lado, el filtro de intervalos se comporta de igual o mejor manera que el filtro de Bloom con Labyrinth y Kmeans. En estos benchmarks existen pocos intervalos de gran tamaño y un número muy reducido de direcciones sueltas (4 % y 2 % respectivamente), lo que hace que el IF no se llene inmediatamente reduciendo así la probabilidad de falso positivo.

El filtro de intervalos funciona para benchmarks con un patrón de acceso a datos específico en el que predominan pocos intervalos de gran tamaño. Sin embargo, no se puede decir que sea una solución general ya que puede resultar dañino para benchmarks que muestran un patrón de acceso aleatorio o con muchos intervalos de pequeño tamaño. Como consecuencia, las siguientes propuestas se centran en los filtros de Bloom, que muestran un comportamiento homogéneo para todo tipo de trazas de direcciones.

A.2. Firma sensible a la localidad

La firma sensible a la localidad (LS-Sig) [79, 78, 80] es una optimización del filtro de Bloom convencional que aprovecha la localidad de referencia de memoria para reducir la probabilidad de falsos positivos. El filtro de Bloom trata de la misma manera tanto los patrones de acceso aleatorios como los patrones que muestran cierta localidad espacial, mapeando las direcciones en bits distintos dentro del filtro. Sin embargo, la firma LS-Sig mapea las direcciones cercanas entre sí compartiendo ciertos bits de manera que se reduce la ocupación del filtro y, por ende, la probabilidad de falsos positivos. Las direcciones que no muestran localidad se mapean de manera convencional.

Un filtro de Bloom mapea un espacio de 2^n direcciones, $N = \{0, 1, ..., 2^n - 1\}$, en un array de 2^m bits (índices), $M = \{0, 1, ..., 2^m - 1\}$, $m \leq n$, por medio de una familia de k funciones de hash, $\{h_0, h_1, ..., h_{k-1}\}$. Las funciones de hash son de la clase H3 [12], ya que ofrecen una distribución de índices de calidad muy cercana a la distribución uniforme. Las funciones de la clase H3 definen una transformación lineal entre una palabra de n bits y otra de m bits: $h_i : GF(2)^{1\times n} \rightarrow GF(2)^{1\times m}$, donde $GF(2)$ es el campo de Galois de dos elementos [104], bajo la función XOR. Se definen dos operaciones básicas en el filtro de Bloom: (i) la inserción de una dirección x, que se realiza poniendo a uno los bits indicados por las funciones de hash ($h_i(x) = 1$), y (ii) chequear que una dirección ha sido insertada, que consiste en comprobar si todos los bits indicados por las funciones de hash están a uno. Sea $BF(x_0, x_1, ..., x_{q-1})$ el conjunto de bits puestos a uno tras insertar una secuencia de q direcciones $x_0, x_1, x_2, ..., x_{q-1}$. Este conjunto viene dado por $\bigcup_{i=0}^{q-1} BF(x_i)$, siendo $BF(x) = \bigcup_{j=0}^{k-1} h_j(x)$.

Un filtro de Bloom cede falsos positivos que pueden venir de dos situaciones. Una dirección y no insertada puede producir un falso positivo si otra dirección x fue insertada en el filtro y $BF(y) = BF(x), y \neq x$. En tal caso, x e y son *alias*, puesto que aplicados a las funciones de hash, los índices resultantes son los mismos. Por otro lado, un falso positivo puede aparecer debido a la *ocupación* del filtro, si $BF(y) \subset BF(x_0, x_1, ..., x_{q-1})$ y la dirección y no es alias de ningún x_i insertado.

Existe una relación entre el número de falsos positivos y el número de funciones hash que se utilizan en un filtro de Bloom. Un número alto de funciones favorece a las transacciones pequeñas, ya que disminuye la probabilidad de alias y, aunque aumenta la ocupación del filtro, al tratarse de transacciones con pocas direcciones, el filtro no se ve afectado en gran medida. Si el número de funciones es pequeño, la ocupación se reduce, por lo que las transacciones grandes se ven favorecidas, aunque existen más alias [92]. Nuestra LS-Sig logra una solución de compromiso manteniendo un k relativamente elevado para favorecer a las transacciones pequeñas, mientras que la sensibilidad a la localidad hace que se inserten menos bits en el filtro reduciendo así la ocupación y favoreciendo a las transacciones grandes.

Haciendo uso de la nomenclatura anterior y basándonos en las definiciones de hashing sensible a localidad o a distancia que se proponen en [16, 47, 51] para formular preguntas acerca de la similaridad entre objetos en espacios métricos, definimos los filtros sensibles a la localidad de la siguiente manera.

Definición A.2.1 Dado un filtro de Bloom que mapea un espacio de 2^n direcciones de memoria, N, en otro espacio de 2^m bits, M, $m \leq n$, a través de una familia de k funciones de hash de la clase H3, y siendo (N, d) y $(\wp(M), d_h)$ dos espacios métricos. Dicho filtro de Bloom se llama (r, δ)-sensible a localidad ((r, δ)-LS), con $r \in \mathbb{N}$ y $\delta : \mathbb{N} \rightarrow \mathbb{N}$, si, para cualquier $x, y \in N$, se cumple que,

- si $1 \leq d(x, y) \leq 2^r - 1$ entonces $0 \leq d_h(BF(x), BF(y)) \leq \delta(d(x, y)) < k$.

■

En un filtro de Bloom diseñado según la Definición A.2.1, posiciones cercanas de memoria se mapean en conjuntos de bits no disjuntos, es decir, comparten ciertos bits del filtro. La función d es la distancia entre dos direcciones dada por la XOR bit a bit, $d(x, y) = x \oplus y$, mientras que la distancia d_h es la mitad de la cardinalidad de la diferencia simétrica de los conjuntos de bits mapeados de cada dirección, $d_h(BF(x), BF(y)) = k - |BF(x) \cap BF(y)|$. El parámetro r actúa como el radio de acción de la LS-Sig. Y la función δ se puede definir dependiente de d, de manera que cuanto más cercanas sean las direcciones, menos disjuntos serán los conjuntos de bits mapeados. Un ejemplo de mapeo LS-Sig se puede ver en la Tabla 5.1, donde $r = k - 1$ y δ se define de manera escalonada.

Para la evaluación experimental de las firmas LS-Sig se exploraron seis firmas diferentes resultantes de combinar dos funciones, δ_0 y δ_1 y tres radios diferentes, 1, 3 y 5. También se propone otra función más homogénea en su comportamiento, δ_P, definida a trozos, que se combina con dos radios diferentes, 3 y 5:

1. (r, δ_0)-LS: Las direcciones comprendidas en intervalos de radio r (i.e. $[0, 2^r - 1], [2^r, 2^{r+1} - 1], \ldots$) se mapean exactamente en los mismos bits del filtro. Lo que significa que 0 índices son diferentes entre los mapas de dichas direcciones:
$$\delta_0(d(x,y)) = 0 \quad \text{si } 1 \leq d(x,y) \leq 2^r - 1.$$

2. (r, δ_1)-LS: Las direcciones comprendidas en intervalos de radio r se mapean en los mismo bits excepto uno. Solamente 1 índice difiere entre mapas:
$$\delta_1(d(x,y)) = 1 \quad \text{si } 1 \leq d(x,y) \leq 2^r - 1.$$

3. (r, δ_P)-LS: Las direcciones comprendidas en intervalos de radio r se mapean dependiendo de la distancia entre dichas direcciones. Así, la función delta se define a trozos de la siguiente manera:
$$\delta_P(d(x,y)) = \begin{cases} 1 & \text{si } d(x,y) = 1 \\ 2 & \text{si } 2 \leq d(x,y) \leq 2^{\lceil \frac{r}{2} \rceil} - 1 \\ 3 & \text{si } 2^{\lceil \frac{r}{2} \rceil} \leq d(x,y) \leq 2^r - 1 \end{cases}$$
Por ejemplo, para $r = 5$, δ_P se define así:
$$\delta_P(d(x,y)) = \begin{cases} 1 & \text{si } d(x,y) = 1 \\ 2 & \text{si } 2 \leq d(x,y) \leq 7 \\ 3 & \text{si } 8 \leq d(x,y) \leq 31 \end{cases}$$

Las Figuras 5.5, 5.6 y 5.7 muestran los resultados obtenidos para δ_0, δ_1 y δ_P respectivamente. Las firmas con δ_0 muestran buenos resultados para tamaños de firma pequeños ya que todos las direcciones en un intervalo se mapean en los mismos bits del filtro, disminuyendo notablemente la ocupación. Sin embargo, cuando la firma es grande, los falsos positivos debidos a la ocupación casi desaparecen y los debidos a los alias hacen que δ_0 rinda peor que las firmas de Bloom genéricas. El rendimiento empeora conforme aumenta el radio ya que los alias aumentan. Con δ_1 al menos un bit del mapa de la dirección es diferente a los de su mismo intervalo, por lo que el efecto de los falsos positivos debidos a alias desaparece en gran medida, aunque ciertos benchmarks como Genome, Intruder y Labyrinth siguen mostrando una ligera caída del rendimiento. Finalmente, las firmas (r, δ_P)-LS-Sig solventan el problema al definir cada función hash con una granularidad distinta. De esta manera, se comporta igual o mejor que las firmas genéricas con firmas de tamaño grande, y es mejor que ellas cuando la firma es pequeña. Las firmas (r, δ_P)-LS-Sig se comportan de manera más homogénea que δ_0 y δ_1 para todos los benchmarks.

A.3. Firmas multiconjunto y asimétrica

Las firmas multiconjunto y asimétrica reconfigurable [77, 82, 83] se proponen para tratar la asimetría que se encuentra con frecuencia en los conjuntos de datos leídos y escritos por las transacciones (véase la cardinalidad de los conjuntos de datos en la Tabla 6.1). Las firmas convencionales disponen de dos filtros de igual tamaño para almacenar las direcciones de la transacción, uno para las direcciones leídas y el otro para las escritas. Por lo tanto, en la mayoría de los casos se hace una utilización ineficiente de la firma de detección de conflictos.

La firma multiconjunto une los filtros de lectura y escritura en un solo filtro de tamaño doble: 2^{m+1}. La Figura 6.1b muestra su implementación. Las funciones de hash del conjunto de lectura y del conjunto de escritura trabajan sobre todo el filtro. Por lo tanto, es necesario que la SRAM que implementa el filtro sea de $2k$ puertos, lo que hace que su tamaño crezca cuadráticamente. Para mitigar dicho aumento de hardware, la firma multiconjunto se puede implementar de forma paralela [92]. El filtro se divide en k subfiltros más pequeños de tamaño $2^{m+1}/k$, de manera que los filtros pasan de tener $2k$ puertos a tener sólo 2, uno para el conjunto de lectura y otro para el de escritura, como se puede ver en el Figura 6.2b. Aún así, una SRAM de doble puerto sigue ocupando más espacio que una SRAM convencional por lo que para reducir la complejidad de las firmas multiconjunto se proponen las firmas de multiconjunto compartido. Estas firmas, como indica la Figura 6.3a, se implementan como una firma multiconjunto en la que s SRAMs, $s \in [0, k]$, poseen un solo puerto, y el resto, $k - s$, siguen siendo de doble puerto. De esta manera, ciertos subfiltros no diferencian entre direcciones leídas y escritas al tener una sola función de hash. Este diseño de firma viene motivado por los datos expuestos en la Figura 6.12, que muestran el porcentaje de direcciones que fueron exclusivamente leídas, exclusivamente escritas, y leídas y a su vez escritas dentro de las transacciones para cada benchmark. Por ejemplo, Bayes y Kmeans muestran un 100 % de direcciones escritas que fueron a su vez leídas. En general, un 30 % de las direcciones accedidas transaccionalmente son leídas y escritas en los benchmarks objeto de estudio. Descifrar el valor adecuado del parámetro s supone un compromiso entre los requisitos de hardware y el rendimiento de la firma. Para valores altos de s cercanos a k la firma tiene más

SRAMs de un solo puerto por lo que se acerca a la firma paralela en cuanto a cantidad de hardware pero los subfiltros pierden la capacidad de discernir entre direcciones leídas o escritas, y vice-versa. En la experimentación exploramos cada posible escenario.

Otra aproximación para solucionar el problema de la asimetría en los conjuntos de datos es nuestra firma asimétrica reconfigurable que se muestra en la Figura 6.3b. La firma reconfigurable parte de una firma paralela convencional como la de la Figura 6.2a, a la que se añade un registro de máscara y una lógica que se utiliza para configurar el número de pares (función de hash, SRAM) que se dedica a cada conjunto de datos. Así, en lugar de tener k subfiltros para el conjunto de lectura y otros k subfiltros para el de escritura, el parámetro a del registro de máscara permite definir la distribución de tales subfiltros entre los conjuntos de datos de manera variable. El registro de máscara se puede instanciar con el valor de a por medio de una instrucción del conjunto de instrucciones, o por el sistema transaccional. En cualquier caso, el problema es encontrar la configuración adecuada para que la firma rinda de la mejor manera posible. Además, se pueden contemplar dos maneras distintas de reconfiguración, una estática por ejecución y otra dinámica por transacción de la ejecución. En esta tesis no se trata este problema, pero se propone un heurístico para su configuración por ejecución dependiendo de las características transaccionales de la aplicación.

Los resultados experimentales de las firmas multiconjunto se muestran en la Figura 6.9. Se puede observar una mejora de rendimiento para Bayes, Genome, Intruder, Vacation y Yada, con un speedup que va de 1,2× a 2,5× dependiendo del benchmark y el tamaño de la firma, tanto para la firma normal como la paralela. La Figura 6.10 muestra el porcentaje de falsos positivos para la firma separada y la multiconjunto. La columna correspondiente a la firma multiconjunto está dividida en falsos positivos y falsos positivos cruzados. Estos últimos resultan del hecho de que el filtro es compartido por las funciones de hash pertenecientes al conjunto de lectura y de escritura, por lo que el llenado del filtro por culpa de las lecturas puede ocasionar falso positivos de escritura y vice-versa. Aún así, el porcentaje total de falsos positivos se mantiene más bajo. Nótese que el filtro multiconjunto ecualiza los falsos positivos de lectura y escritura. Por otro lado, Kmeans y Labyrinth se comportan peor con las firmas multiconjunto para tamaños de firma pequeños. Estos benchmarks generan muchos falsos positivos cruzados en la firma y también muestran una cardinalidad media elevada de sus conjuntos de lectura y escritura (véase la Tabla 6.1), lo que hace que la ocupación del filtro exceda en muchos casos el umbral de 2/3 a partir del cual se ha visto que la firma multiconjunto empieza a ceder más falsos positivos que la firma separada.

Para mejorar el rendimiento de las firmas multiconjunto y a su vez reducir sus requisitos hardware se estudian las firmas de multiconjunto compartido, cuyos resultados se pueden ver en la Figura 6.13, en la que se exploran todos los posibles valores del parámetro s. A medida que crece s, los resultados mejoran. De hecho, con $s = 4$ se obtienen los mejores resultados, excepto para Bayes y Genome, cuyo rendimiento baja entorno a un 25 % con respecto a la firma separada para un tamaño de 8Kbit. Por lo tanto, de manera conservadora y generalista, se recomienda el uso de la firma de multiconjunto compartido con $s = 3$, que se comporta como la separada para tamaños de filtro alto, mientras que obtiene muy buenos resultados para los filtros de tamaño menor. Por último, se utilizó el hashing sensible a la localidad, propuesto en la sección anterior, para mejorar los resultados de la firma de multiconjunto compartido con $s = 3$. Existen dos implementaciones posibles. En la primera, L1, el subfiltro que no comparte las funciones de hash tiene dichas funciones trabajando con máxima granularidad, es decir, las funciones del subfiltro que es capaz de diferenciar entre direcciones leídas y escritas es sensible a la localidad con máximo radio. En la segunda implementación, L2, esas funciones de hash trabajan de manera normal, sin sensibilidad a la localidad, mientras que son los subfiltros compartidos los que obtienen la mejora de la localidad. Los resultados obtenidos se muestran en la Figura 6.16, que muestra un rendimiento parecido para ambas soluciones, siendo ligeramente mejor la variante L2, y mejorando ambas al esquema sin la mejora de localidad.

En cuanto a la firma asimétrica reconfigurable, como se ha visto con anterioridad, puede ser reconfigurada dinámicamente en tiempo de ejecución, en base a las características de cada transacción. Sin embargo, en nuestro caso se ha escogido una configuración estática por ejecución completa del benchmark para la evaluación del filtro. Concretamente, como el conjunto de datos de lectura de la transacción suele ser igual o mayor que el de escritura, se experimentó con tres valores para el parámetro de configuración: $a = 5$, que implica un conjunto de lectura con 5 SRAMs, por 3 del conjunto de escritura; $a = 6$, que dedica 6 subfiltros al conjunto de lectura; y $a = 7$, con una sola SRAM para el conjunto de escritura. Nótese que la firma asimétrica reconfigurable con $a = 4$ equivale a la firma simétrica convencional. Los resultados obtenidos se muestran en la Figura 6.11. Para cada benchmark, existe una configuración de la firma asimétrica que obtiene un rendimiento parecido al de la firma multiconjunto o incluso mejor, en el caso de Labyrinth y Kmeans, que ya se comportaban peor con la firma multiconjunto que con la convencional. Sin embargo, la firma asimétrica puede no ser una solución general si carecemos de los medios para obtener el valor de configuración adecuado. Aquí se propone el uso de un heurístico con el que se obtiene una configuración estática por ejecución que nos proporciona un rendimiento relativamente bueno, aunque no óptimo, el cual vendría dado por una configuración por transacción. Se trata de la razón entre la media de las cardinalidades de los conjuntos de lectura de las transacciones y la media de las cardinalidades de los conjuntos de escritura de las mismas. Esta media se puede ver en la última columna de la Tabla 6.1. Si se redondea dicha razón a la razón más cercana de las posibles configuraciones de la firma asimétrica, i.e. $a = 4$ implica razón 1, $a = 5$ razón 1.67, $a = 6$ razón 3 y $a = 7$ razón 7, se puede obtener la mejor configuración estática para la aplicación. Así, Bayes tiene una razón de 1.88 que es más cercana a la razón de la configuración $a = 5$, y obtiene los mejores resultados con tal configuración. Genome tiene una razón de 2.88 y obtiene los mejores resultados con la configuración $a = 6$. El heurístico es válido para todos los benchmarks excepto para Intruder y Vacation, que tienen una razón de 7.64 y 5.47 respectivamente, que deberían ejecutarse mejor con $a = 7$, sin embargo, dan mejores resultados con $a = 6$, aunque la diferencia no es muy acusada.

Por último, se estimó el consumo de energía, tiempo y área de las firmas objeto de estudio y se obtuvieron los valores de la Tabla 6.3. En resumen, la firma de multiconjunto compartido con $s = 3$ necesita 1.2 veces más hardware, es un 12 % más lenta y consume un 6 % más energía que la firma paralela debido a la SRAM de doble puerto.

A.4. Estudio de escalabilidad

Para explorar la escalabilidad de las firmas propuestas en esta tesis se utilizó el benchmark EigenBench [46], especialmente desarrollado para analizar características ortogonales de sistemas transaccionales. Dicho benchmark se modificó para incluir la generación de trazas de direcciones de memoria con cierta localidad espacial dada por parámetro. Se exploró la respuesta de las firmas a la contención del sistema, el tamaño de la transacción y la concurrencia.

Con respecto a la contención, que es definida como la probabilidad de conflicto de una transacción, se analizaron dos configuraciones de EigenBench que se pueden ver en la Tabla 7.1. Una con transacciones cortas y otra con transacciones largas, que a su vez se probó con tres valores diferentes del parámetro de localidad espacial. La Figura 7.3a muestra los resultados para la configuración con transacciones cortas. Se muestra el speedup con respecto a la versión secuencial y la versión paralela no protegida por mecanismos de sincronización como límite máximo de paralelismo. Se puede observar que todas las variantes de las firmas se comportan de igual manera ya que el tamaño de la firma es mucho mayor al de la transacción (4Kbit en este caso), por lo que nuestras propuestas no dañan la ejecución de transacciones pequeñas. También se puede observar que la alta contención hace que el sistema no escale. Las demás gráficas de la Figura 7.3 corresponden a transacciones largas con valores crecientes de localidad. En este caso, nuestras propuestas mejoran el rendimiento de la firma paralela separada, y para el caso de mayor localidad se llega a igualar el rendimiento de la firma perfecta. Nótese que demasiada contención puede llevar al sistema HTM a ofrecer un rendimiento peor que la versión secuencial.

La Figura 7.4 muestra los resultados del estudio del tamaño de la transacción. Existen dos gráficas, una para una situación de transacciones con conjuntos de lectura y escritura del mismo tamaño, y otra para una configuración asimétrica. El tamaño de transacción varía de 40 direcciones hasta 640. Las firmas perfectas ofrecen resultados muy similares a la versión no protegida del benchmark, sin embargo, las firmas imperfectas, a medida que aumenta el tamaño de la transacción, van produciendo falsos positivos que deterioran el rendimiento hasta el punto de hacer que el rendimiento sea similar al de la versión secuencial. Nuestras propuestas de firma mitigan en cierta medida esa bajada de rendimiento para valores medios de tamaño de transacción, pero tienden a rendir como las firmas paralelas convencionales a medida que crece el tamaño de la transacción.

La Figura 7.5 muestra los resultados del estudio de concurrencia en los que se puede ver que nuestras propuestas escalan mejor que la firma convencional. También se puede apreciar que aumentando el número de núcleos se empeora el efecto de los falsos conflictos de las firmas imperfectas, por lo que será necesario seguir mejorando dichas firmas para los futuros multiprocesadores.

A.5. Conclusiones y trabajo futuro

En la era de los multiprocesadores en un chip, existe un creciente interés en encontrar nuevos paradigmas de programación que faciliten la escritura de programas concurrentes. La memoria transaccional surge como una alternativa a la programación convencional con cerrojos con el fin de facilitar dicha tarea y para sacar el máximo provecho a los CMPs. De hecho, la memoria transaccional hardware está ganando popularidad hasta tal punto que los principales fabricantes de procesadores están trabajando para incluir extensiones transaccionales en sus nuevos modelos de multiprocesador. Esta tesis se enmarca en el contexto de la memoria transaccional hardware, campo en el que se proponen varias optimizaciones del mecanismo de detección de conflictos, una parte crítica para el buen funcionamiento del sistema. Concretamente, nuestras propuestas se centran en nuevas implementaciones de las firmas transaccionales, que pueden ser clave en la virtualización del sistema.

El primer esquema de firma propuesto en esta tesis es el *filtro de intervalos*. Se trata de una alternativa a las firmas basadas en filtros de Bloom, que almacena las direcciones accedidas por la transacción en forma de intervalos. El filtro de intervalos logra superar el rendimiento de las firmas Bloom convencionales siempre que las aplicaciones muestren un patrón de acceso a datos con localidad de tal manera que las direcciones se puedan disponer en unos pocos intervalos de direcciones contiguas. Este es el caso de algunos benchmarks de la suite STAMP como Kmeans y Labyrinth. Sin embargo, esta firma alternativa no consigue mejorar el rendimiento de las firmas con filtro de Bloom cuando el patrón de acceso forma muchos intervalos pequeños o muestra grandes cantidades de accesos aleatorios, ya que se implementa como una memoria asociativa con un número reducido de entradas para mantener un tamaño asequible. Como consecuencia, el filtro de intervalos no es una solución general pero podría ser una opción a tener en cuenta en caso de que se sepa de antemano el patrón de acceso de la aplicación.

La siguiente contribución de esta tesis sigue la línea de aprovechar la localidad, pero esta vez se basa en los filtros de Bloom, que nos proporcionan una solución más general que la que se consigue con los filtros de intervalos. Se trata de las *firmas sensibles a la localidad*, que definen nuevos mapas para las funciones hash de los filtros que reducen el número de bits que se insertan en estos. Así, direcciones de memoria cercanas entre sí comparten ciertos bits del filtro de Bloom, y como resultado, los falsos conflictos se reducen para aquellas transacciones que muestran localidad espacial, mientras que no se afecta a las transacciones que no muestran dicha localidad. Se proponen varias formas de la firma sensible a localidad que rinden de igual o mejor manera que las firmas convencionales. Sin embargo, encontramos que las versiones δ_P, que definen funciones a trozos con diferente granularidad por hash, son mejores que el resto puesto que se comportan de manera más uniforme independientemente del benchmark. Por último, como las firmas sensibles a la localidad se basan en nuevos mapas para las funciones hash, su implementación no requiere hardware extra, e incluso se puede ahorrar hasta un 3 % en puertas XOR.

Las siguientes contribuciones de esta tesis tratan de solventar el problema de la asimetría en los conjuntos de lectura y escritura de las transacciones. Las firmas convencionales disponen de filtros separados de igual tamaño para mantener los conjuntos de direcciones. Sin embargo, las transacciones muestran con frecuencia cierta disparidad en la cardinalidad de dichos conjuntos, lo que introduce ineficiencias en el uso de las firmas. La primera optimización que se propone en este contexto se ha llamado *firma multiconjunto*. Se utiliza un solo filtro de Bloom con doble puerto para mantener los dos conjuntos de direcciones, tanto el de las lecturas como el de las escrituras, por lo que los falsos positivos tienden a igualarse en ambos conjuntos. Se encuentra una mejora debido al mejor aprovechamiento del filtro, excepto para las transacciones cuyo tamaño excede un cierto umbral. Puesto que las memorias de doble puerto necesitan más hardware para su implementación, se propone una mejora que hemos llamado *firma compartida multiconjunto* que hace que ciertas partes del filtro sólo tengan una función compartida entre los conjuntos de lectura y escritura. Los resultados, incluyendo la mejora de la localidad antes mencionada, pueden mejorar a las firmas convencionales hasta en un 47 % de media, puesto que una gran cantidad de direcciones accedidas dentro de las transacciones se leen y se escriben al mismo tiempo.

Otra aproximación para el tratamiento de la asimetría en los conjuntos de datos transaccionales es nuestra *firma asimétrica reconfigurable*. Se parte de una firma paralela convencional con cada filtro dividido en subfiltros, uno por función hash. Así, y por medio de un registro de máscara de configuración, se pueden asignar los diferentes subfiltros para que formen parte de la firma de lectura o de escritura, pudiendo tener una firma de lectura más grande, igual o más pequeña que la de escritura. La configuración se puede realizar por transacción o por aplicación. Nosotros proponemos una configuración por aplicación que implica un profiling de la ejecución para obtener la razón resultante de dividir la cardinalidad media del conjunto de lectura y de escritura de las transacciones. Con este heurístico se obtiene un buen resultado de las firmas asimétricas reconfigurables, aunque no sea la solución óptima.

Por último, se muestra un estudio comparativo de la escalabilidad de nuestras soluciones con respecto a la convencional en términos de la contención del sistema, el tamaño de la transacción y el número de transacciones concurrentes. Para ello se ha usado un nuevo benchmark para el estudio de características ortogonales en sistemas de memoria transaccional, llamado EigenBench, que ha sido modificado para incluir el modelado de localidad de referencia espacial. Los resultados obtenidos muestran que la contención puede dañar seriamente la ejecución de código transaccional, hasta el punto de que la versión paralela sea más lenta que la versión serie. Nuestras propuestas amortiguan la degradación del rendimiento, sobretodo cuando la aplicación muestra cierta cantidad de localidad espacial. Sin embargo, tanto la firma perfecta como las imperfectas convergen, limitando así la posibilidad de mejora en el desarrollo de firmas imperfectas. Con respecto al tamaño de la transacción, nuestras propuestas de firma mejoran los resultados de la firma convencional en la mayoría de los casos. Sin embargo, tienden a igualar el tiempo de ejecución de la firma convencional conforme crece el tamaño de la transacción, que daña seriamente la ejecución. Los resultados también sugieren que los sistemas de memoria transaccional implícitos, aunque más fáciles de programar, podrían llevar a una mala configuración de las firmas asimétricas si dicha configuración se realiza partiendo de los parámetros de la aplicación en lugar de profiling, ya que muchas variables no transaccionales se toman como transaccionales en estos sistemas. También se puede observar que el efecto de los falsos conflictos en las firmas imperfectas aumenta con el número de transacciones concurrentes, aunque nuestras mejoras hacen que el sistema escale mejor.

En esta tesis se proponen una serie de optimizaciones para las firmas de detección de conflictos que ayudan en la virtualización del sistema transaccional. Sin embargo, existe trabajo por hacer en este campo. Para terminar, se apuntan los siguientes caminos de trabajo futuro:

- Nuestra firma asimétrica reconfigurable ha sido evaluada con una configuración por aplicación dando buenos resultados. Sin embargo, vale la pena explorar la reconfiguración por transacción, puesto que el tamaño de los conjuntos de lectura y escritura puede cambiar de una transacción a otra dentro de la misma aplicación. Por un lado, se puede utilizar profiling para obtener la razón entre el conjunto de lectura y el de escritura de cada transacción y configurar la firma en el comienzo de las mismas. Por otro lado, se puede explorar una solución en la que el sistema transaccional reconfigure la firma basándose en información obtenida durante la ejecución, como la historia de transacciones previas o estadísticas de la firma en el aborto.

- Ciertos resultados obtenidos en esta tesis exponen el problema de la alta contención, que puede llevar a la versión transaccional a ejecutarse más lenta que la versión serie. En [9] se estudian las patologías que pueden afectar al rendimiento de los sistemas de memoria transaccional hardware, y esta línea de investigación sigue abierta con varios trabajos que están al tanto del problema de la contención [102, 73]. En el contexto de las firmas para detección de conflictos, se podría estudiar cómo detectar si la firma está cediendo demasiados falsos positivos, y proceder a la posterior serialización de la ejecución para evitar la pérdida de rendimiento.

- Los sistemas transaccionales implícitos insertan direcciones innecesarias en la firma, como variables privadas, que incrementan el número de falsos positivos. Yen et al. [111] y Sanyal et al. [93] atajan el problema proporcionando nuevas construcciones al programador para reservar memoria privada o compartida. Pero el programador tiene que extremar las precauciones ya que un error podría provocar la incorrección del programa. Por otro lado, los sistemas transaccionales explícitos pueden hacer que la programación sea compleja [88], lo que contradice el principal propósito de la memoria transaccional. La detección automática de accesos a variables privadas podría ser una línea de investigación para la mejora del rendimiento de las firmas mientras que se mantiene la programabilidad.

- Los sistemas transaccionales cuya detección de conflictos se basa en firmas con filtros de Bloom no pueden confiar en las firmas para decidir si los bloques de memoria escritos en la transacción tienen que ser almacenados en el log, puesto que un falso positivo puede causar que no se almacene un bloque que debería ser almacenado. Estos sistemas necesitan un filtro para el log que mantenga los bloques que han sido recientemente almacenados en el log para evitar que se almacenen repetidamente. Estudiar el efecto en el rendimiento del filtro de log sería interesante, así como estudiar una posible ayuda de la firma de detección de conflictos.

La memoria transaccional es un paradigma prometedor para simplificar el desarrollo de aplicaciones paralelas. No esta claro lo que le depara el futuro, pero la industria del hardware ya está apostando por ella. Esta tesis es una pequeña contribución a la memoria transaccional, con la esperanza de que ayude a clarificar su futuro.

Bibliography

[1] S.V. Adve and K. Gharachorloo. Shared memory consistency models: a tutorial. *Computer*, 29(12):66–76, 1996. (Cited on page 28)

[2] A.R. Alameldeen and D.A. Wood. Variability in architectural simulations of multi-threaded workloads. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, HPCA'03, pages 7–18, 2003. (Cited on page 42)

[3] C.S. Ananian, K. Asanovic, B.C. Kuszmaul, C.E. Leiserson, and S. Lie. Unbounded transactional memory. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, HPCA'05, pages 316–327, 2005. (Cited on pages 4, 23, 24 and 142)

[4] J.L. Baer and W.H. Wang. On the inclusion properties for multi-level cache hierarchies. In *Proceedings of the 15th Annual International Symposium on Computer architecture*, ISCA'88, pages 73–80, 1988. (Cited on page 42)

[5] L. Baugh, N. Neelakantam, and C. 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, 2008. (Cited on page 25)

[6] B.H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970. (Cited on pages 4 and 28)

[7] C. Blundell, J. Devietti, E.C. Lewis, and M.M.K. Martin. Making the fast case common and the uncommon case simple in unbounded transactional memory. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA'07, pages 24–34, 2007. (Cited on pages 4 and 5)

[8] J. Bobba, N. Goyal, M.D. Hill, M.M. Swift, and D.A. Wood. TokenTM: Efficient execution of large transactions with hardware transactional memory. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA'08, pages 127–138, 2008. (Cited on pages 4, 5 and 24)

[9] J. Bobba, K.E. Moore, H. Volos, L. Yen, M.D. Hill, M.M. Swift, and D.A. Wood. Performance pathologies in hardware transactional memory. In *Proceedings of the 34th annual international symposium on Computer architecture*, ISCA'07, pages 81–91, 2007. (Cited on pages 4, 24, 121, 138 and 156)

[10] P. Bose, H. Guo, E. Kranakis, A. Maheshwari, P. Morin, J. Morrison, M. Smid, and Y. Tang. On the false-positive rate of Bloom filters. *Information Processing Letters*, 108(4):210–213, 2008. (Cited on pages 28 and 30)

[11] A. Broder and M. Mitzenmacher. Network applications of Bloom filters: A survey. *Internet Mathematics*, 1(4):485–509, 2004. (Cited on page 53)

[12] L. Carter and M. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18(2):143–154, 1979. (Cited on pages 31, 97 and 146)

[13] C. Cascaval, C. Blundell, M. Michael, H.W. Cain, P. Wu, S. Chiras, and S. Chatterjee. Software transactional memory: Why is it only a research toy? *Queue*, 6(5):46–58, 2008. (Cited on page 17)

[14] L. Ceze, J. Tuck, C. Cascaval, and J. Torrellas. Bulk disambiguation of speculative threads in multiprocessors. In *Proceedings of the 33th Annual International Symposium on Computer Architecture*, ISCA'06, pages 227–238, 2006. (Cited on pages 4, 23, 27, 53, 93 and 143)

[15] L. Ceze, J. Tuck, P. Montesinos, and J. Torrellas. BulkSC: Bulk enforcement of sequential consistency. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA'07, pages 278–289, 2007. (Cited on page 28)

[16] M.S. Charikar. Similarity estimation techniques from rounding algorithms. In *Proceedings of the 34th Annual ACM Symposium on Theory of Computing*, STOC'02, pages 380–388, 2002. (Cited on pages 36, 65 and 147)

[17] S. Chaudhry, R. Cypher, M. Ekman, M. Karlsson, A. Landin, S. Yip, H. Zeffer, and M. Tremblay. Rock: A high-performance sparc CMT processor. *IEEE Micro*, 29:6–16, 2009. (Cited on pages 4 and 25)

[18] B. Chazelle, J. Kilian, R. Rubinfeld, and A. Tal. The Bloomier filter: An efficient data structure for static support lookup tables. In *Proceedings of the 15th annual ACM-SIAM symposium on Discrete algorithms*, SODA'04, pages 30–39, 2004. (Cited on page 37)

[19] W. Choi and J. Draper. Locality-aware adaptive grain signatures for transactional memories. In *Proceedings of the IEEE International Symposium on Parallel and Distributed Procesing*, IPDPS'10, pages 1–10, 2010. (Cited on pages 35 and 80)

[20] W. Choi and J. Draper. Implementation of unified signatures for transactional memory systems. In *Proceedings of the 54th IEEE International Midwest Symposium on Circuits and Systems*, MWSCAS'11, pages 1–4, 2011. (Cited on page 35)

[21] W. Choi and J. Draper. Unified signatures for improving performance in transactional memory. In *Proceedings of the IEEE International Symposium on Parallel Distributed Processing Symposium*, IPDPS'11, pages 817–827, 2011. (Cited on pages 35 and 104)

[22] D. Christie, J.W. Chung, S. Diestelhorst, M. Hohmuth, M. Pohlack, C. Fetzer, M. Nowack, T. Riegel, P. Felber, P. Marlier, and E. 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, 2010. (Cited on pages 4 and 26)

[23] W. Chuang, S. Narayanasamy, G. Venkatesh, J. Sampson, M. Van Biesbrouck, G. Pokam, O. Colavin, and B. Calder. Unbounded page-based transactional memory. In *Proceedings of the 12th International Conference on Architectural Support for Programming Language an Operating Systems*, ASPLOS'06, pages 347–358, 2006. (Cited on page 4)

[24] J. Chung, H. Chafi, C.C. Minh, A. McDonald, B.D. Carlstrom, C. Kozyrakis, and K. Olukotun. The common case transactional behavior of multithreaded programs. In *Proceedings of the 12th International Symposium on High Performance Computer Architecture*, HPCA'06, pages 266–277. 2006. (Cited on pages 5 and 65)

[25] S. Cohen and Y. Matias. Spectral Bloom filters. In *Proceedings of the ACM SIGMOD international conference on Management of data*, SIGMOD'03, pages 241–252, 2003. (Cited on page 37)

[26] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. In *Proceedings of the 12th International Conference on Architectural Support for Programming Language an Operating Systems*, ASPLOS'06, pages 336–346, 2006. (Cited on pages 4 and 142)

[27] P.J. Denning. The locality principle. Technical report, Naval Postgraduate School, Monterey, California, 2008. (Cited on page 71)

[28] D. Dice, Y. Lev, M. Moir, and D. Nussbaum. Early experience with a commercial hardware transactional memory implementation. In *Proceedings of the 14th International Conference on Architectural Support for Programming Language an Operating Systems*, ASPLOS'09, pages 157–168, 2009. (Cited on page 4)

[29] D. Dice, O. Shalev, and N. Shavit. Transactional locking ii. In *Proceedings of the 20th International Symposium on Distributed Computing*, (DISC'06), pages 194–208, 2006. (Cited on pages 4, 17, 124 and 142)

[30] K.P. Eswaran, J.N. Gray, R.A. Lorie, and I.L. Traiger. The notions of consistency and predicate locks in a database system. *Communications of the ACM*, 19(11):624–633, 1976. (Cited on page 17)

[31] L. Fan, P. Cao, J. Almeida, and A.Z. Broder. Summary cache: A scalable wide-area web cache sharing protocol. *IEEE/ACM Transactions on Networking*, 8(3):281–293, 2000. (Cited on page 37)

[32] P. Felber, C. Fetzer, and T. Riegel. Dynamic performance tuning of word-based software transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, PPoPP'08, pages 237–246, 2008. (Cited on page 17)

[33] Free Software Foundation. GNU General Public License. http://www.gnu.org/copyleft/gpl.html. (Cited on page 41)

[34] K. Fraser and T. Harris. Concurrent programming without locks. *ACM Transactions on Computer Systems*, 25(2), May 2007. (Cited on page 18)

[35] D. Geer. Industry trends: Chip makers turn to multicore processors. *IEEE Computer*, 38(5):11–13, 2005. (Cited on pages 2 and 141)

[36] R. Guerraoui, M. Herlihy, and B. Pochon. Polymorphic contention management. In *Proceedings of the 19th international conference on Distributed Computing*, DISC'05, pages 303–323, 2005. (Cited on page 16)

[37] L. Hammond, V. Wong, M. Chen, B.D. Carlstrom, J.D. Davis, B. Hertzberg, M.K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. In *Proceedings of the 31th Ann. International Symposium on Computer Architecture*, ISCA'04, pages 102–113, 2004. (Cited on pages 4, 23 and 142)

[38] F. Hao, M. Kodialam, T.V. Lakshman, and H. Song. Fast dynamic multiple-set membership testing using combinatorial bloom filters. *IEEE/ACM Transactions on Networking*, 20(1):295–304, 2012. (Cited on page 36)

[39] T. Harris and K. Fraser. Language support for lightweight transactions. *SIGPLAN Notices*, 38(11):388–402, October 2003. (Cited on pages 4, 18 and 142)

[40] T. Harris, J.R. Larus, and R. Rajwar. *Transactional Memory, 2nd edition.* Morgan & Claypool Publishers, 2010. (Cited on pages 2, 3 and 142)

[41] T. Harris, M. Plesko, A. Shinnar, and D. Tarditi. Optimizing memory transactions. *SIGPLAN Notices*, 41(6):14–25, 2006. (Cited on pages 4, 17, 19 and 142)

[42] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, 1991. (Cited on page 4)

[43] M. Herlihy, V. Luchangco, and M. Moir. A flexible framework for implementing software transactional memory. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, OOPSLA'06, pages 253–262, 2006. (Cited on page 19)

[44] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the 22nd annual symposium on Principles of distributed computing*, PODC'03, pages 92–101, 2003. (Cited on pages 4, 12, 18, 48 and 142)

[45] M. Herlihy and J.E.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, 1993. (Cited on pages 3, 4, 19, 21, 142 and 143)

[46] S. Hong, T. Oguntebi, J. Casper, N. Bronson, C. Kozyrakis, and K. Olukotun. Eigenbench: A simple exploration tool for orthogonal TM characteristics. In *Proceedings of the IEEE International Symposium on Workload*

Characterization, IISWC'10, pages 1–11, 2010. (Cited on pages 121, 125, 127 and 152)

[47] P. Indyk and R. Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *Proceedings of the 30th annual ACM symposium on Theory of computing*, STOC'98, pages 604–613, 1998. (Cited on pages 36, 65 and 147)

[48] S.A.R. Jafri, M. Thottethodi, and T.N. Vijaykumar. LiteTM: Reducing transactional state overhead. In *Proceedings of the 16th International Symposium on High-Performance Computer Architecture*, HPCA'10, pages 81–92, 2010. (Cited on page 4)

[49] M. Jimeno, K.J. Christensen, and A. Roginsky. Two-tier Bloom filter to achieve faster membership testing. *Electronics Letters*, 44(7):503–504, 2008. (Cited on page 53)

[50] C. Kim, D. Burger, and S.W. Keckler. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. In *Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, ASPLOS-X, pages 211–222, 2002. (Cited on page 42)

[51] A. Kirsch and M. Mitzenmacher. Distance-sensitive bloom filters. In *Proceedings of the 8th Workshop on Algorithm Engineering and Experiments*, ALENEX'06, pages 41–50, 2006. (Cited on pages 36, 65 and 147)

[52] I. Kotera, R. Egawa, H. Takizawa, and H. Kobayashi. Modeling of cache access behavior based on Zipf's law. In *Proceedings of the 9th workshop on memory performance: dealing with applications, systems and architecture*, pages 9–15, 2008. (Cited on pages 71 and 124)

[53] S. Kumar, M. Chu, C.J. Hughes, P. Kundu, and A. Nguyen. Hybrid transactional memory. In *Proceedings of the 11th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP'06, pages 209–220, 2006. (Cited on pages 4, 25 and 142)

[54] J.R. Larus and R. Rajwar. *Transactional Memory.* Morgan & Claypool Publishers, 2007. (Cited on pages 3 and 142)

[55] Y. Lev, M. Moir, and D. Nussbaum. PhTM: Phased transactional memory. In *Proceedings of the 2nd Workshop on Transactional Computing*, TRANSACT'07, 2007. (Cited on page 25)

[56] J.L. Lo, J.S. Emer, H.M. Levy, R.L. Stamm, D.M. Tullsen, and S.J. Eggers. Converting thread-level parallelism to instruction-level parallelism via simultaneous multithreading. *ACM Transactions on Computer Systems*, 15(3):322–354, 1997. (Cited on pages 2 and 141)

[57] C.K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V.J. Reddi, and K. Hazelwood. PIN: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM SIGPLAN Conference on Programming Languages Design and Implementation*, PLDI'05, pages 190–200, 2005. (Cited on pages 39, 45 and 72)

[58] M. Lupon, G. Magklis, and A. Gonzalez. FASTM: A log-based hardware transactional memory with fast abort recovery. In *Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques*, PACT'09, pages 293–302, 2009. (Cited on page 25)

[59] M. Lupon, G. Magklis, and A. Gonzalez. A dynamically adaptable hardware transactional memory. In *Proceedings of the 43rd IEEE/ACM Annual International Symposium on Microarchitecture*, MICRO'10, pages 27–38, 2010. (Cited on pages 4, 28 and 143)

[60] P.S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, B. Werner, and B. Werner. Simics: A full system simulation platform. *IEEE Computer*, 35(2):50–58, 2002. (Cited on pages 39 and 145)

[61] A. Marowka. Back to thin-core massively parallel processors. *Computer*, 44(12):49–54, 2011. (Cited on pages 2 and 141)

[62] M. Martin, C. Blundell, and E. Lewis. Subtleties of transactional memory atomicity semantics. *IEEE Computer Architecture Letters*, 5(2):17–, 2006. (Cited on page 12)

[63] M.M.K. Martin, D.J. Sorin, B.M. Beckmann, M.R. Marty, M. Xu, A.R. Alameldeen, K.E. Moore, M.D. Hill, and D.A. Wood. Multifacet's general execution-driven multiprocessor simulator GEMS toolset. *ACM SIGARCH Computer Architecture News*, 33(4):92–99, 2005. (Cited on pages 39 and 145)

[64] M. Matsumoto and T. Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation*, 8(1):3–30, 1998. (Cited on page 51)

[65] M. Mehrara, J. Hao, P.C. Hsu, and S. Mahlke. Parallelizing sequential applications on commodity hardware using a low-cost software transactional memory. In *Proceedings of the ACM SIGPLAN Conference on Programming Languages Design and Implementation*, PLDI'09, pages 166–176, 2009. (Cited on pages 4, 28 and 143)

[66] C.C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford Transactional Applications for Multi-Processing. In *Proceedings of the IEEE International Symposium on Workload Characterization*, IISWC'08, pages 35–46, 2008. (Cited on pages 46, 63 and 145)

[67] C.C. Minh, M. Trautmann, J. Chung, A. McDonald, N. Bronson, J. Casper, C. Kozyrakis, and K. 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, 2007. (Cited on pages 4, 25, 28, 91 and 143)

[68] G.E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8):114–117, 1965. (Cited on pages 1 and 141)

[69] K.E. Moore, J. Bobba, M.J. Moravan, M.D. Hill, and D.A. Wood. LogTM: Log-based transactional memory. In *Proceedings of the 12th International Symposium on High-Performance Computer Architecture*, HPCA'06, pages 254–265, 2006. (Cited on pages 4, 24, 142 and 143)

[70] M.J. Moravan, J. Bobba, K.E. Moore, L. Yen, M.D. Hill, B. Liblit, M.M. Swift, and D.A. Wood. Supporting nested transactional memory in LogTM. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, ASPLOS-XII, pages 359–370, 2006. (Cited on page 24)

[71] J.E.B. Moss and A.L. Hosking. Nested transactional memory: model and architecture sketches. *Science of Computer Programming*, 63(2):186–201, 2006. (Cited on page 11)

[72] N. Muralimanohar, R. Balasubramonian, and N. Jouppi. CACTI 6.0: A tool to model large caches. Technical Report HPL-2009-85, HP Laboratories, 2009. (Cited on page 117)

[73] A. Negi, P. Stenstrom, R. Titos-Gil, M.E. Acacio, and J.M. Garcia. Pi-TM: Pessimistic invalidation for scalable lazy hardware transactional memory. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, PACT'11, pages 203–204, 2011. (Cited on pages 25, 138 and 156)

[74] K. Olukotun and L. Hammond. The future of microprocessors. *Queue*, 3(7):26–29, 2005. (Cited on pages 1 and 141)

[75] L. Orosa, E. Antelo, and J.D. Bruguera. FlexSig: Implementing flexible hardware signatures. *ACM Transactions on Architecture and Code Optimization*, 8(4):30:1–30:20, 2012. (Cited on page 35)

[76] S.M. Pant and G.T. Byrd. Extending concurrency of transactional memory programs by using value prediction. In *Proceedings of the 6th ACM conference on Computing Frontiers*, CF'09, pages 11–20, 2009. (Cited on page 25)

[77] R. Quislant, E. Gutierrez, O. Plata, and E.L. Zapata. Hardware signature designs to deal with asymmetry in transactional data sets. *IEEE Transactions on Parallel and Distributed Systems*, In Press, DOI:10.1109/TPDS.2012.138. (Cited on pages 6, 91, 144 and 149)

[78] R. Quislant, E. Gutierrez, O. Plata, and E.L. Zapata. LS-Sig: Locality-sensitive signatures for transactional memory. *IEEE Transactions on Computers*, In Press, DOI:10.1109/TC.2011.230. (Cited on pages 6, 63, 144 and 146)

[79] R. Quislant, E. Gutierrez, O. Plata, and E.L. Zapata. Improving signatures by locality exploitation for transactional memory. In *Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques*, PACT'09, pages 303–312, 2009. (Cited on pages 6, 63, 75, 93, 144 and 146)

[80] R. Quislant, E. Gutierrez, O. Plata, and E.L. Zapata. Signatures and locality in transactional memory. In *Proceedings of the international summer school on advanced computer architecture and compilation for embedded systems*, ACACES'09, pages 93–96, 2009. (Cited on pages 6, 63, 144 and 146)

[81] R. Quislant, E. Gutierrez, O. Plata, and E.L. Zapata. Interval filter: a locality-aware alternative to bloom filters for hardware membership queries by interval classification. In *Proceedings of the 11th international conference on Intelligent data engineering and automated learning*, IDEAL'10, pages 162–169, 2010. (Cited on pages 6, 53 and 144)

[82] R. Quislant, E. Gutierrez, O. Plata, and E.L. Zapata. Multiset signatures for transactional memory. In *Proceedings of the 25th international conference on Supercomputing*, ICS'11, pages 43–52, 2011. (Cited on pages 6, 91, 144 and 149)

[83] R. Quislant, E. Gutierrez, O. Plata, and E.L. Zapata. Unified locality-sensitive signatures for transactional memory. In *Proceedings of the 17th international conference on Parallel processing*, EURO-PAR'11, pages 326–337, 2011. (Cited on pages 6, 91, 144 and 149)

[84] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing transactional memory. In *Proceedings of the 32th Annual International Symposium on Computer Architecture*, ISCA'05, pages 494–505, 2005. (Cited on pages 4, 22 and 142)

[85] H.E. Ramadan, C.J. Rossbach, and E. Witchel. Dependence-aware transactional memory for increased concurrency. In *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*, MICRO'08, pages 246–257, 2008. (Cited on page 25)

[86] M. V. Ramakrishna, E. Fu, and E. Bahcekapili. Efficient hardware hashing functions for high performance computers. *IEEE Transactions on Computers*, 46(12):1378–1381, 1997. (Cited on page 31)

[87] J. Reinders. Transactional synchronization in Haswell. Intel's software blogs. http://software.intel.com/en-us/blogs/2012/02/07/transactional-synchronization-in-haswell/, 2012. (Cited on pages 4 and 26)

[88] C.J. Rossbach, O.S. Hofmann, and E. Witchel. Is transactional programming actually easier? In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP'10, pages 47–56, 2010. (Cited on pages 138 and 157)

[89] B. Saha, A. Adl-Tabatabai, R.L. Hudson, C.C. Minh, and B. Hertzberg. McRT-STM: A high performance software transactional memory system for a multi-core runtime. In *Proceedings of the 11th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP'06, pages 187–197, 2006. (Cited on page 17)

[90] B. Saha, A.-R. Adl-Tabatabai, and Q. Jacobson. Architectural support for software transactional memory. In *Proceedings of the 39st IEEE/ACM Annual International Symposium on Microarchitecture*, MICRO'06, pages 185–196, 2006. (Cited on pages 4, 25 and 142)

[91] D. Sanchez. Design and implementation of signatures for transactional memory systems. Technical Report CS-TR-2007-1611, University of Wisconsin-Madison, 2007. (Cited on page 75)

[92] D. Sanchez, L. Yen, M.D. Hill, and K. Sankaralingam. Implementing signatures for transactional memory. In *Proceedings of the 40th Annual*

IEEE/ACM International Symposium on Microarchitecture, MICRO'07, pages 123–133, 2007. (Cited on pages 28, 31, 33, 34, 54, 59, 65, 69, 75, 89, 93, 121, 144, 147 and 149)

[93] S. Sanyal, S. Roy, A. Cristal, O. S. Unsal, and M. Valero. Dynamically filtering thread-local variables in lazy-lazy hardware transactional memory. *High Performance Computing and Communications, 10th IEEE International Conference on*, 0:171–179, 2009. (Cited on pages 138 and 156)

[94] W.N. Scherer, III and M.L. Scott. Advanced contention management for dynamic software transactional memory. In *Proceedings of the 24th annual ACM symposium on Principles of distributed computing*, PODC'05, pages 240–248, 2005. (Cited on page 16)

[95] N. Shavit and D. Touitou. Software transactional memory. In *Proceedings of the 14th annual ACM symposium on Principles of distributed computing*, PODC'95, pages 204–213, 1995. (Cited on pages 4, 17 and 142)

[96] A. Shriraman, M. Spear, H. Hossain, V. Marathe, S. Dwarkadas, and M. Scott. An integrated hardware-software approach to flexible transactional memory. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA'07, pages 104–115, 2007. (Cited on page 25)

[97] S. Shriraman, S. Dwarkadas, and M. Scott. Flexible decoupled transactional memory support. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA'08, pages 139–150, 2008. (Cited on pages 4, 25, 28 and 143)

[98] T. Skare and C. Kozyrakis. Early release: Friend or foe? In *Workshop on Transactional Memory Workloads*. 2006. (Cited on pages 12 and 48)

[99] D. Thiebaut, J.L. Wolf, and H.S. Stone. Synthetic traces for trace-driven simulation of cache memories. *IEEE Transactions on Computers*, 41:388–410, 1992. (Cited on page 124)

[100] S. Thoziyoor, N. Muralimanohar, J.H. Ahn, and N.P. Jouppi. CACTI 5.1. Technical Report HPL-2008-20, HP Laboratories, 2008. (Cited on pages 59 and 145)

[101] R. Titos, M.E. Acacio, and J.M. García. Directory-based conflict detection in hardware transactional memory. In *Proceedings of the 15th international conference on High performance computing*, HiPC'08, pages 541–554, 2008. (Cited on page 35)

[102] R. Titos, A. Negi, M.E. Acacio, J.M. García, and P. Stenstrom. ZEBRA: A data-centric, hybrid-policy hardware transactional memory design. In *Proceedings of the international conference on Supercomputing*, ICS'11, pages 53–62, 2011. (Cited on pages 25, 138 and 156)

[103] S. Tomic, C. Perfumo, C. Kulkarni, A. Armejach, O. Cristal, T. Harris, and M. Valero. EazyHTM: Eager-lazy hardware transactional memory. In *Proceedings of the 42nd IEEE/ACM Annual International Symposium on Microarchitecture*, MICRO'09, pages 145–155, 2009. (Cited on page 4)

[104] H. Vandierendonck and K. De Bosschere. XOR-based hash functions. *IEEE Transactions on Computers*, 54(7):800–812, 2005. (Cited on pages 31, 64 and 146)

[105] M.M. Waliullah and P. Stenstrom. Efficient management of speculative data in hardware transactional memory systems. In *Proceedings of the International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation*, SAMOS'08, pages 158 –164, 2008. (Cited on page 25)

[106] M.M. Waliullah and P. Stenstrom. Schemes for avoiding starvation in transactional memory systems. *Concurrency and Computation: Practice and Experience*, 21(7):859–873, 2009. (Cited on page 24)

[107] D.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, 1991. (Cited on pages 2 and 141)

[108] S. Wang, W. Xu, Z. Pang, D. Wu, Q. Dou, and X. Yang. DTM: Decoupled hardware transactional memory to support unbounded transaction and operating system. In *Proceedings of the 38th International Conference on Parallel Processing*, ICPP'09, pages 228–236, 2009. (Cited on page 4)

[109] S.J.E. Wilton and N.P. Jouppi. CACTI: an enhanced cache access and cycle time model. *IEEE Journal of Solid-State Circuits*, 31(5):677 –688, 1996. (Cited on page 117)

[110] L. Yen, J. Bobba, M.R. Marty, K.E. Moore, H. Volos, M.D. Hill, M.M. Swift, and D.A. Wood. LogTM-SE: Decoupling hardware transactional memory from caches. In *Proceedings of the 13th International Symposium on High-Performance Computer Architecture*, HPCA'07, pages 261–272, 2007. (Cited on pages 4, 24, 28, 41, 43, 101 and 143)

[111] L. Yen, S.C. Draper, and M.D. Hill. Notary: Hardware techniques to enhance signatures. In *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO'08, pages 234–245, 2008. (Cited on pages 5, 35, 69, 86, 89, 97, 119, 138 and 156)

[112] C. Zilles and R. Rajwar. Transactional memory and the birthday paradox. In *Proceedings of the 19th annual ACM symposium on Parallel algorithms and architectures*, SPAA'07, pages 303–304, 2007. (Cited on page 33)