Multiset Signatures for Transactional Memory

Ricardo Quislant, Eladio Gutierrez, Oscar Plata and Emilio L. Zapata
Dept. of Computer Architecture
University of Malaga, Spain
{quislant, eladio, oplata, zapata}@uma.es

ABSTRACT

Transactional Memory (TM) systems must record the memory locations read and written (read and write sets) by concurrent transactions in order to detect conflicts. Some TM implementations use signatures for this purpose, which summarize read and write sets in bounded hardware at the cost of false positives (detection of non-existing conflicts).

Read/write signatures are usually implemented as two separate Bloom filters with the same size. In contrast, transactions usually exhibit read/write sets of uneven cardinality, where read sets use to be larger than write sets. Thus, the read filter populates earlier than the write one and, consequently the read signature false positive rate may be high while the write filter has still a low occupation.

In this paper, a multiset signature design is proposed which records both the read and write sets in the same Bloom filter without adding significant hardware complexity. Several designs of multiset signatures are analyzed and evaluated. New problems arise related to hardware complexity and the existence of cross false positives, i.e. new false positives coming from the fact that both sets share the same filter. Additionally, multiset signatures are enhanced using locality-sensitive hashing, proposed by the authors in a previous work. Experimental results show that the multiset approach is able to reduce the false positive rate and improve the execution performance in most of the tested codes, without increasing the required hardware area in a noticeable amount.

Keywords

Hardware transactional memory, signatures, bloom filters, H3 hashing, memory locality

1. INTRODUCTION

With the development of the single-chip parallel processors, known as CMPs (Chip Multiprocessors), manycore or multicore processors [10], with an internal architecture similar to a shared-memory multiprocessor, the common programmer is forced to deal with the multithreaded parallel programming model to extract the maximum performance from the cores. In general, writing multithreaded programs is a fairly complex task that poses a major obstacle to exploit multicore processors. Parallelism introduces non-determinism that must be controlled by a careful design of the computational threads and their coordination through explicit synchronization. Shared data used in critical sections must be accessed in mutual exclusion to avoid race conditions. Lock-based techniques are traditionally used to provide mutual exclusion by serializing the execution of concurrent threads in critical sections. By narrowing these sections (finer lock granularity) the thread serialization may be minimized, but at a cost of increasing the lock overhead and the risk of deadlock. In addition, locks have other disadvantages, like convoying or priority inversion, problems difficult to detect and solve. Finally, locks lack of effective mechanisms for abstraction and composition.

Transactional Memory (TM) [14, 12] emerges as an alternative to the conventional multithreaded programming to ease the writing of concurrent programs. TM introduces the concept of transaction that allows semantics to be separated from implementation. A transaction is a block of computations that appears to be executed with atomicity and isolation. Thus, transactions replace a pessimistic lock-based model by an optimistic one and solve the abstraction and composition problems.

TM systems execute transactions in parallel, committing non-conflicting ones. A conflict occurs when a memory location is accessed by several concurrent transactions and at least one access is a write.

In this paper we are mainly interested on hardware implementations of TM, that include those systems (HTM) that provide most of the required TM mechanisms implemented in hardware, at the core level [11, 2, 7, 22, 19, 9, 28], as well as those systems that provide hardware support to speed up parts of a software TM implementation (STM) [24, 27, 26]. These TM systems must record all memory reads and writes during the execution of transactions in order to detect conflicts.

Signatures have been recently proposed to store the memory locations of such memory reads and writes, that are called read and write sets, respectively. Examples of systems that use signatures are BulkSC [6], LogTM-SE [30], SigTM [18], FlexTM [26], and STMlite [17].

These systems implement signatures as per-thread Bloom filters [3]. Basically, they use fixed hardware (except for STMlite, that uses a software implementation) to summarize

an unbounded amount of read and write memory locations at the cost of false positives (i.e. detection of non-existing conflicts).

Read/write signatures are usually implemented as two separate Bloom filters with the same size. In contrast, transactions usually exhibit read/write sets of uneven cardinality, where read sets use to be larger than write sets. As a result, the signature for reads populates much more than the one for writes and, consequently, the false positive rate for the read signature may be high while, at the same time, the write filter has still a low occupation, with negligible false positive rate. This situation could be handled by an asymmetric resizing of the signatures. However, this solution suffers from lack of generality as resizing would depend on the specific application. In addition, hardware implementation restrictions of signatures introduce important complications to this approach. For instance, signature size may be restricted to a power of two, dynamic resizing introduces important additional hardware costs, feedback from the programmer or the compiler may be required, and so on.

In this paper a different approach is followed. A multiset signature design is proposed which records both the read and write sets in the same Bloom filter without adding significant hardware complexity. When sharing the filter, both the read and the write false positive rates can be equalized. Several designs of multiset signatures are analyzed and evaluated. New problems arise related to hardware complexity and the existence of cross false positives, i.e. new false positives coming from the fact that both sets share the same filter. Additionally, as a second contribution, multiset signatures are enhanced using locality-sensitive hashing, proposed by the authors in a previous work [21]. Finally, as a third contribution, the proposed multiset and locality-sensitive multiset signatures were implemented in the Wisconsin GEMS LogTM-SE simulator [16], in order to evaluate their performance, and in CACTI [29] in order to evaluate the hardware area and energy requirements. Experimental results show that the multiset approach is able to reduce the false positive rate and improve the execution performance in most of the tested codes, without increasing the required hardware area in a noticeable amount and slightly increasing the power consumption.

The rest of the paper is organized as follows. In next section we present a background on signatures, describing how they are usually designed and implemented. A brief review of the related work is discussed. In Section 3 we introduce our proposed multiset signature design, discussing its basics, how they are implemented, and a comparison with other signature designs. Section 4 shows an analysis of our proposed signatures and determines false positive rates in different contexts. Next, Section 5 presents the implementation of multiset signatures on the GEMS simulator, and discusses how our novel signature design may improve the execution performance in several cases. In addition, an analysis of area and energy requirements using CACTI is also shown. Finally, Section 6 concludes the paper.

2. BACKGROUND AND RELATED WORK

In the context of TM, each concurrent thread uses its signatures to record all the memory locations issued when executing inside a transaction. These locations are sorted out into a read set (RS) and a write set (WS). Thus, each thread needs a pair of private signatures. As they are used for conflict detection amongst concurrent transactions, signatures do not tolerate false negatives (undetected true conflicts) but may assume a limited amount of false positives (false conflicts). On the other hand, the RS and WS sizes are unknown in advance, therefore, signatures should not limit the number of addresses to be tracked. In addition, test and insertion of an address should be fast operations.

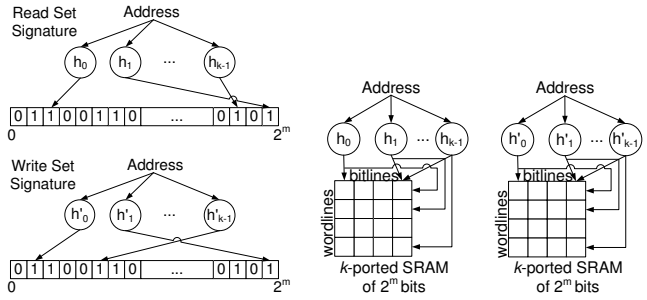
Fulfilling the requirements above, Ceze et al. [7] proposed a signature implementation with per-thread Bloom filters. These filters were devised to test whether an element is a member of a set in a time and space-efficient way. The Bloom filter allows insertions of an unbounded number of elements at the cost of false positives, but not false negatives (elements can be added to the set, but not removed). It comprises a bit array and k different hash functions that map elements into k randomly distributed bits of the array. At first, all the array bits are set to 0. Inserting an element into the Bloom filter consists in setting to 1 the k bits given by the hash functions. Test for membership consists in checking that those k bits are asserted.

Bloom filters are also known as true or regular Bloom filters. Sanchez et al. [25] proposed the parallel Bloom filter as an alternative hardware-efficient implementation of regular Bloom filters. Whereas the regular filter is implemented as a k-ported SRAM, the parallel one consists of k 1-ported SRAMs, yielding the same or better false positives rate. Next section shows the implementation of both filters, regular and parallel. The same work concludes that Bloom filters should include H_3 class hash functions [5], instead of bit-selection hash functions [23], since they are closer to random distribution. However, H_3 hashings are hardware expensive and need an XOR tree per hash bit.

An alternative hardware-efficient implementation of hash functions, Page-Block-XOR hashing (PBX), has been proposed in [31]. They use the concept of entropy to find input bits to the hash functions with high randomness, allowing to reduce the hardware complexity of those functions. Notary also proposes a technique to reduce the number of asserted bits in the signature, based on segregating addresses into private and shared sets. Then, only the shared addresses are recorded in the signature. 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.

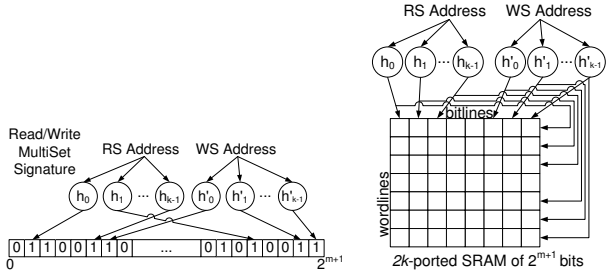
Recently, Choi et al. [8] proposed adaptive grain signatures, that keep the history of transaction aborts and dynamically changes the input bit range to the hash functions based on the abort history. The aim of this design is to reduce the number of false positives that harm the execution performance.

Signatures have been adopted by several HTM systems instead of keeping transactional state in the cache memory. Modifying caches to track transactional information have been proved to pose major constraints into TM 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. Signatures are used to enforce sequential consistency in BulkSC [6]. LogTM-SE [30] uses them in the directory and it ensures paging and context switching with global signatures. SigTM [18] is similar to LogTM-SE. VTM [22] uses a global signature for cache victimization. Finally, signatures were also



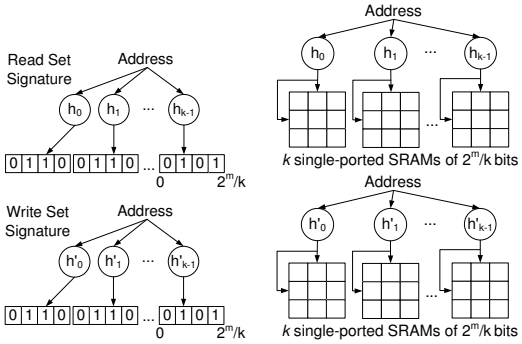
(a) Design (b) Hardware implementation

Figure 1: Regular Bloom filters.



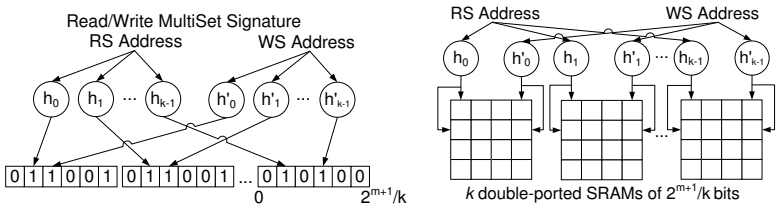
(a) Design (b) Hardware implementation

Figure 2: Regular MultiSet Bloom filters.



(a) Design (b) Hardware

Figure 3: Parallel Bloom filters.



(a) Design (b) Hardware implementation

Figure 4: Parallel MultiSet Bloom filters.

used for conflict detection in FPGA-based HTM systems [13].

3. MULTISET SIGNATURE DESIGNS

This section introduces the design and implementation of the different multiset (MS) signature proposals. Regular and parallel schemes [25] are shown along with its MS versions. Also, a novel and more efficient signature is proposed called parallel multiset shared signature.

3.1 Regular Multiset Signatures

A regular Bloom filter consists of one 2^m bit array and k independent hash functions that index the array. Two operations are provided, insertion and test for membership. Initially, the bit array is set to zero. To insert an address into the filter, first, the address passes through the k hash functions giving k values or indexes. Then those bits in the bit array indexed by the k hash values are set to one. On the other hand, to test for an address the first step is analogous to the insertion operation, that is, compute the hash values for the address. Next, the k bits indexed by such values have to be checked. If they are all set to 1 then the test is positive, otherwise the address is not in the filter.

Figure 1 shows the design and implementation of regular signatures. Two Bloom filters are needed, one for the read set and the other one for the write set (see Figure 1a). They are usually of the same length, 2^m bits. Regarding the hardware, a regular Bloom filter can be implemented as an SRAM. The bit array can be divided into words, thus memory cells are placed forming a matrix with wordlines as rows and bitlines as columns, as shown in Figure 1b. This way, the address given by the hash function is divided into word address and bit address. In case of $k > 1$ hash function filters SRAMs should be k-ported since they can perform the operations in parallel in one cycle instead of operating in several cycles. However, multi-ported memories require more hardware than single-ported ones and signatures must keep both concise and fast.

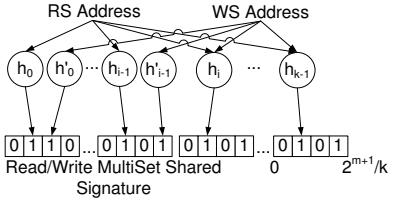
Regular multiset signatures merge the read set and the write set into the same filter of 2^{m+1} bits. Figure 2 shows the design and implementation of this proposal. Now, read set hash functions, $h_0...h_{k-1}$, and write set hash functions, $h'_0...h'_{k-1}$ share the filter. The main drawback of regular MS signatures is that they must be implemented by duplicating the number of ports of the SRAM and it translates into a quadratic growth of the required area. In order to save in area and also to maintain time-efficiency, parallel signatures were used [25] [7] that do not need multi-ported SRAMs.

3.2 Parallel Multiset Signatures

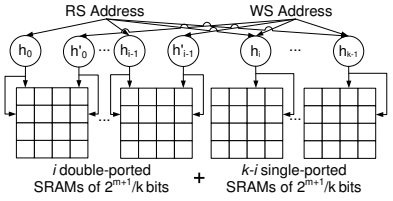
A parallel Bloom filter consists of k arrays of $2^m/k$ bits. Each hash function only indexes its own array, therefore one bit is set into each array on insertion.

Figure 3 shows the design an implementation of parallel Bloom signatures. They comprise two separate parallel filters to keep track of read set and write set addresses. 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 proven to yield similar or better performance than regular ones [25] [21].

On the other hand, Figure 4 depicts the design and im-



(a) Design



(b) Hardware implementation

Figure 5: Parallel MultiSet Shared Bloom filters.

plementation of the multiset counterpart for the parallel signature. In this case, the bit array is also partitioned into k smaller arrays but of $2^{m+1}/k$ bits. Now, each array is indexed by two hash functions, one for the read set, h_i, and the other one for the write set, h'_i. Therefore, parallel multiset filters need 2-ported SRAMs instead of single-ported ones taking about twice the area of parallel Bloom filters. To alleviate this problem, parallel multiset shared signatures have been proposed.

3.3 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 (see Section 5.4). In such a case, storing the same address twice may be redundant but the filter must be able to discriminate whether the address was only read or also written. Figure 5 shows the proposed solution. The signature is a parallel multiset signature with some arrays indexed by only one hash function. This way, to insert an address into the filter, some arrays do not take into account if the address is either read or written, they simply record one bit representing the address. However, the rest of the arrays discriminate between reads and writes since they are addressed by a read hash function h and a write hash function h'. Consequently, in case of an insertion to the write set, h' would set a bit in those arrays. Then, if the same address is subsequently inserted to the read set, a different bit would be set in the same arrays. Therefore, certain read and write hash functions are said to be *shared*.

Figure 5b shows how the parallel multiset shared signature is implemented. Unlike parallel multiset signatures, only i double-ported SRAMs are needed, the rest remains single-ported. Thus, if i is low, parallel MS shared signatures do not require much more area than parallel signatures.

Finally, hash functions are implemented as H_3 XOR hash functions [5] that only 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 [31].

4. MULTISET SIGNATURE ANALYSIS

The following expression is commonly assumed to calculate the false positive probability for a single filter of 2^m bits [25] [21], after inserting a sequence of s elements using k hash functions:

$$p_{\text{FP}}(m,k,s) = \left(1 - \left(1 - \frac{1}{2^m}\right)^{sk}\right)^k. \quad (1)$$

First, let us consider two different conventional Bloom filters that store the read and write sets separately. A first goal in this section is determining the average false positive rate of the system consisting of these two separate read/write filters. With this purpose, let us suppose an arbitrary address sequence of length s to be inserted in the filters. For such a sequence, let p_R be the probability of an address being exclusively read, p_W the probability of an address being exclusively written and p_{RW} the probability of an address being both read and written ($p_R + p_W + p_{RW} = 1$). Therefore, the false positive probability in each filter can be expressed as:

$$\begin{aligned} p_{\text{FP}}^{read}(m,k,s) &= \left(1 - \left(1 - \frac{1}{2^m}\right)^{s(p_R+p_{RW})k}\right)^k, \\ p_{\text{FP}}^{write}(m,k,s) &= \left(1 - \left(1 - \frac{1}{2^m}\right)^{s(p_W+p_{RW})k}\right)^k. \end{aligned} \quad (2)$$

In order to get the mathematical expectation of the false positive rate in this two-filter system, it is necessary to define the probability that both filters (read and write) are checked. By denoting with c_R and c_W such probabilities respectively, we can determine the average false positive rate of the two separate filters as:

$$\begin{aligned} \overline{p}_{FP}^{\text{SEP}}(m,k,s) = E[p_{\text{FP}}(m,k,s)] = \\ c_R p_{\text{FP}}^{read}(m,k,s) + c_W p_{\text{FP}}^{write}(m,k,s) \end{aligned} \quad (3)$$

Note that in real transactions the number of writes is usually lower than the number of reads and consequently it is expected that the false positive rate will be higher for the reads. Additionally, those addresses being read and written will be inserted in both filters, causing for both an increasing of occupancy.

In general, the checking pattern will depend on how the threads check each of the filters searching for possible data dependencies. Such a pattern is unknown until run-time and very dependent on the programming strategy used when designing the parallel multithreaded code. Other issues as the coherence protocol and how thread transactions may be aborted or resumed may also have influence. This checking pattern will determine the two weight factors c_R, c_W appearing in the previous expression.

If both filters are joined in a single one of double size, where both read and write sets are inserted, expression 1 remains valid since reads and writes use a different set of k hashing functions. Thus, the false positive probability for this multiset scheme will be:

$$p_{FP}^{\text{MULTI}}(m,k,s) = p_{\text{FP}}(m+1,k,s(1+p_{RW})) \quad (4)$$

Observe that the addresses that are both read and written are inserted using different hashing functions, they are also inserted twice in the multiset filter.

Two different scenarios have been depicted in Figures 6 and 7, where equations 3 and 4 have been evaluated. In the first case it is considered that the checking pattern is

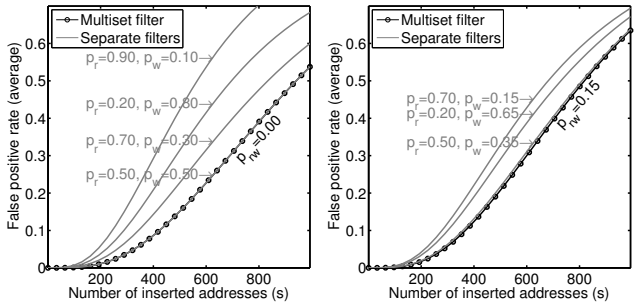


Figure 6: Evaluation of the false positive rate average considering that inserting and checking patterns are the same. Separate filters and multiset configuration are compared. $k = 4$, $2^m = 1Kbit$.

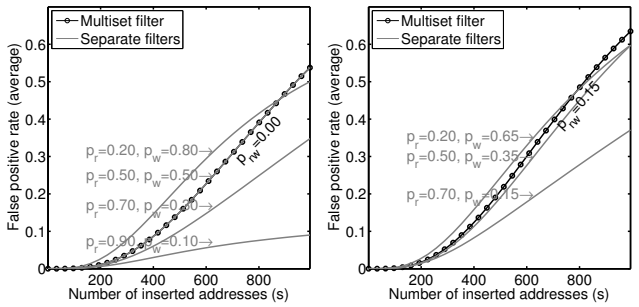


Figure 7: Evaluation of the false positive rate average considering that the read filter is checked only for writes, but the write filter is checked both for writes and reads. Separate filters and multiset configuration are compared. $k = 4$, $2^m = 1Kbit$.

the same than that of insertion in each filter. This corresponds to weight factors $c_R = \frac{p_R+p_{RW}}{p_R+2*p_{RW}+p_W}$ and $c_W = \frac{p_{RW}+p_W}{p_R+2*p_{RW}+p_W}$. On the other hand, Figure 7 has been generated from equation 3 assuming that when a read dependence is checked the write filter is probed but when checking a write dependence, both filters, read and write, need to be checked. Under this observation, if we suppose that the checking of other threads follows the same probabilities that the inserting thread, we can estimate that $c_R = \frac{p_{RW}+p_W}{p_R+2p_{RW}+2p_W}$ and $c_W = \frac{p_R+p_{RW}+p_W}{p_R+2p_{RW}+2p_W}$. Observe that if the number of addresses both read and written increases, the analyzed configurations (separate filters and multiset) show a trend to be very similar in their expected false positive rate. Actually, for values close to $p_{RW} = 1$ both configurations yield the same rate (not shown). Nevertheless, for low values of p_{RW} the behavior varies greatly with the checking pattern. In such cases, if the number of checks are equalized between the two separated filters (Figure 6) the multiset proposal can perform with less false positives.

5. EXPERIMENTAL EVALUATION

This section describes the simulation environment and methodology (Section 5.1), the STAMP benchmark suite used to evaluate our proposal (Section 5.2) and the experimental results obtained from the simulator (Sections 5.3, 5.4 and 5.5). Finally, hardware requirements are analyzed in Section 5.6.

5.1 Simulation environment

The simulation environment comprises a full system execution-driven simulator called Simics [15] along with the HTM module GEMS [16] provided by the Wisconsin Multifacet Project as open-source.

On one hand, Simics simulates the SPARC architecture and it is able to run an unmodified copy of a Solaris operating system. Solaris 10 was installed on the simulated machine and all workloads run on top of it. On the other hand, GEMS's Ruby module implements the LogTM-SE HTM [30] and also includes a detailed timing model for the memory system. Ruby was modified to include the proposed multiset signature designs described in Section 3.

The base CMP system consists of 16 in-order, single-issue cores. Each core has a 32KB split, 4-way associative, 64B block private L1 cache. L2 cache is unified, 8MB capacity, 16-bank, 8-way associative, and 64B block size. A packet-switched interconnect with 64B links connects the cores and cache banks. Cache coherence implements the MESI protocol and maintains an on-chip directory which holds a bit vector of sharers. Main memory is 4GB.

Simulation experiments use perfect signatures (no false positives, hardware unimplementable) as the reference. Filter size ranges from 64 bits, which matches the word length in SPARC architecture, to 8K bits length, which matches the performance of perfect signatures for the simulated benchmarks, to gain a comprehensive insight into multiset signatures behavior. All filters use 4 hash functions of the H_3 family [5] since they have been proved to be better than using less hash functions and other low-quality hash family like bit-selection [25]. Same H_3 matrices of Ruby were used.

Finally, Ruby adds pseudorandom delays to the latency of memory accesses to deal with variability in simulation experiments. Therefore, multiple runs of each experiment have been done to obtain confident error bars [1].

5.2 Workloads

All the workloads used in this paper belong to the Stanford's STAMP suite [4]. 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 since they put the most pressure in signatures. STAMP workloads have been adapted to GEMS by applying Luke Yen's patches from the University of Wisconsin, Madison.

Specifically, the patches introduce the following changes to the benchmarks: (i) every thread is bound to a processor to keep the operating system from descheduling it; (ii) a per-thread memory pool is used instead of "malloc" to allocate dynamic data; (iii) these memory pools are traversed before starting computation in order to avoid page faults inside transactions; (iv) shared data structures are padded to avoid false sharing at cache line level; (v) library functions used inside transactions are also called before entering transactions to let the linker fill in the Procedure Linkage Table; (vi) some transactions in Vacation benchmark have been split to improve scalability for small signatures; (vii) in Labyrinth benchmark, the code that privatizes the grid is enclosed in an open transaction. Regarding change (ii), a modified version of the memory pool library provided by

Table 1: Workloads: Input parameters and TM characteristics

Bench	Input	#xact	Time in xact	avg $\|RS\|$	avg $\|WS\|$	max $\|RS\|$	max $\|WS\|$
Bayes	-v32 -r1024 -n2 -p20 -s0 -i2 -e2	523	94%	76.9	40.9	2067	1613
Genome	-g512 -s64 -n8192	30304	86%	12.1	4.2	400	156
Intruder	-a10 -l128 -n128 -s1	12123	96%	19.1	2.5	267	20
Kmeans	-m40 -n40 -t0.05 -i rand-n1024-d1024-c16	1380	6%	99.7	48.5	134	65
Labyrinth	-i rand-x32-y32-z3-n64	158	100%	76.5	62.9	278	257
SSCA2	-s13 -i1.0 -u1.0 -l3 -p3	47295	19%	2.9	1.9	3	2
Vacation	-n4 -q60 -u90 -r16384 -t4096	24722	97%	19.7	3.6	90	30
Yada	-a20 -i 633.2	5384	100%	62.7	38.4	776	510

STAMP was used.

Table 1 summarizes the input parameters and main transactional characteristics of the benchmarks. Column "#xact" shows the number of committed transactions. Column "Time in xact" lists the percentage of execution cycles of the benchmark staying inside transactions. The last columns stand for the average and the maximum values of RS and WS size distributions in cache lines.

5.3 Regular and Parallel Multiset Signature Results

In this section a comparison between regular/parallel signatures and regular/parallel multiset signatures is conducted.

Figure 8 shows the results obtained for every benchmark in the suite. The y axis represents the time in cycles that has been normalized to that of the perfect signatures. Light gray lines depict multiset signatures and dark gray lines depict non-multiset ones. Solid line is regular and dashed line is parallel. The x axis represents the size of the filter. For example, the 64bit value means that non-multiset signatures have been configured to use two 64bit filters, one for the RS and the other one for the WS, while multiset signatures use only one 128bit filter. Let's see that benchmarks can be classified into three behavioral groups:

1. SSCA2: This benchmark is no signature dependent because of its small transactions, the smallest of the whole suite as can be seen in Table 1. 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 normal ones.

   Bayes and Yada get a slightly 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, if one transaction reads lots of memory locations the filter will be filled with ones. Then, if no write has been introduced but a test for a write occurs, it could yield a cross false positive, because of filter occupancy, if such a test checks four bits that have been set to 1 by several reads.

   On the other hand, Genome, Intruder and Vacation perform better using multiset signatures. Genome is 1.4× faster with filters of 1Kbit and 2Kbit, Intruder also exhibits about 1.4× speedup from 256Kbit filter downwards, and up to 2.5× of speedup is achieved for Vacation. These three benchmarks show relatively small transactions in average (see Table 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 signatures for Kmeans but parallel MS ones perform worse for some filter sizes. In the case of Labyrinth, multiset signatures perform much worse than non-multiset filters specially for parallel ones. Labyrinth's transactions are large in average and many cross false positives come up because of occupancy of the filter. Next sections propose certain configuration enhancements that will ameliorate filter occupancy.

Parallel multiset signatures perform similar than general ones for most cases as shown in Figure 8 and require much less area (see Section 5.6). Therefore, subsequent optimizations have been explored using the parallel scheme.

5.4 Parallel Multiset Shared Signature Results

Parallel multiset shared signature design and implementation were described in Section 3.3. The motivation behind such a signature comes from Figure 9 which shows the percentage of addresses that have been both read and written inside transactions for each benchmark. Three bars can be seen per benchmark. From left to right: (i) the number of addresses read and written with respect to the total number of addresses read; (ii) the number of addresses read and written with respect to the total number of addresses written; and (iii) the number of addresses read and written with respect to the total number of addresses (without repetition). Let us see that more than 50% of locations written are also read for most of the benchmarks. For example, Bayes, Kmeans and Yada exhibit close to 100% of written addresses that have been also read. Overall, about 30% of total locations addressed by each benchmark has been both read and written.

Given that the percentage of addresses both read and written inside transactions is not inconsiderable, next step is to figure out the number of hash functions that can be shared by read set and write set in multiset signatures without losing performance. For that purpose, experiments were conducted where shared functions ranges 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 10 shows the execution time of parallel multiset shared signatures. As read set and write set hash functions are shared the results get better for all the benchmarks.

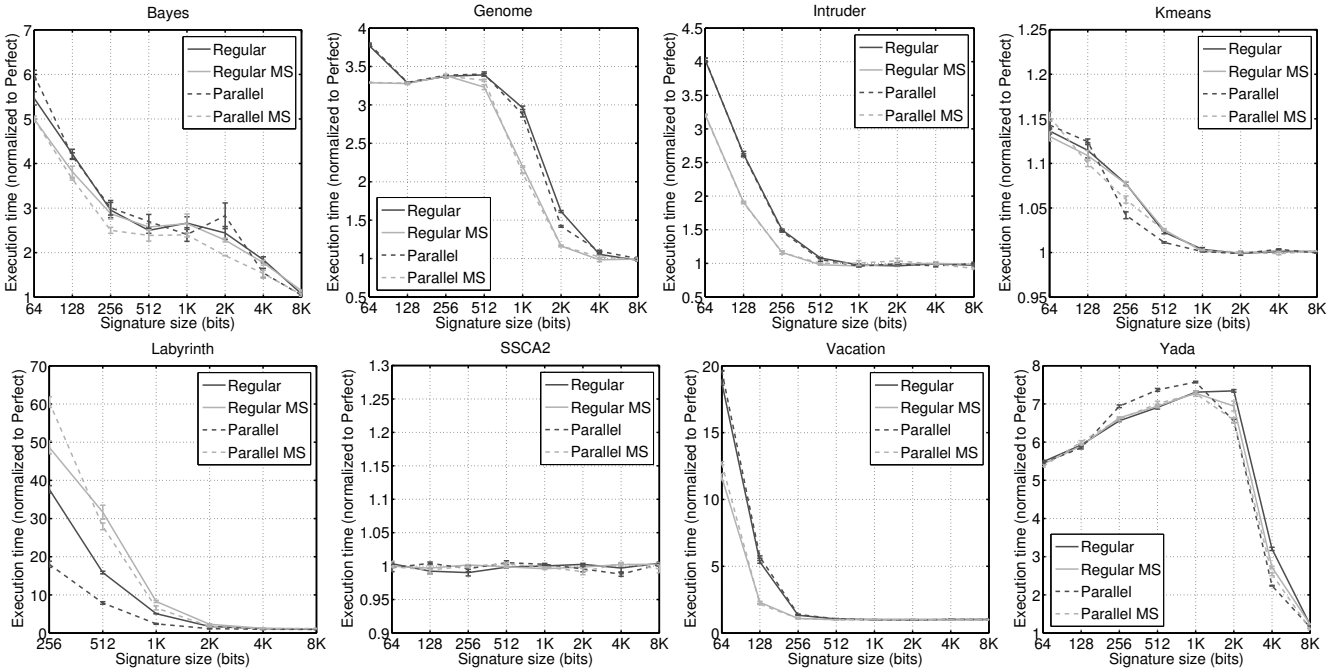


Figure 8: Execution time normalized to perfect signatures (no false positives) comparing regular and parallel signatures to regular and parallel multiset ones.

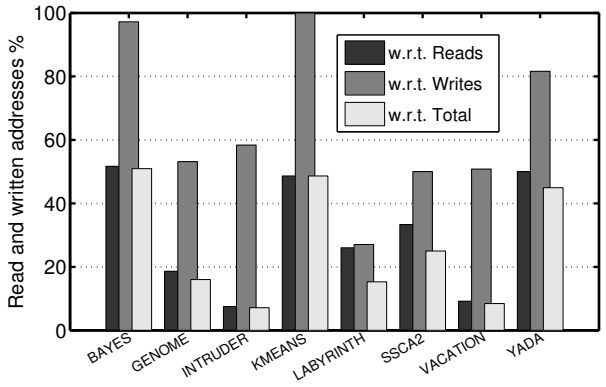


Figure 9: Percentage of addresses that have been both read and written inside transactions

In fact, MS shared 4 gets the best results for every workload except Bayes and Genome, which execution is slowed down about 1.25× with respect to parallel filters for 8Kbit signatures. Therefore, parallel multiset shared 3 signatures should be used instead of shared 4 since they perform equal or better than parallel filters for all signature sizes, while shared 4 signatures perform slightly better than shared 3 for small signature sizes but perform worse than parallel ones for large signatures.

5.5 Parallel Multiset Shared Locality Signature Results

In this section, locality-sensitive hashing [21] is used to enhance parallel multiset shared 3 signatures described in Section 5.4.

Locality-sensitive hashing is able to take advantage of locality of reference, a property which every application exhibits in some extent, to store an address stream more concisely. In a Bloom filter with locality-sensitive hash functions, nearby locations assert non-disjoint bits into the bit array saving in occupancy. Table 2 shows how the locality hash functions operate. Let's see that, for contiguous addresses, the number of hashing outputs with different values is 1. Addresses with distance 2 are different in no more than 2 hashing outputs and, addresses with distance greater than $2^{k-1} - 1$ may have no hashing outputs in common.

Figure 11 shows the results of combining parallel multiset shared 3 signatures with locality-sensitive hashing. Two different combinations are shown:

- Loc1: Shared 3 signatures share hash functions h_1, h_2 and h_3 while h_0 and h'_0 functions remains separate (see Section 3.3) and assert different bits in the same filter, ones for the read set and others for the write set. Locality 1 scheme makes that h_0 and h'_0 behave as h_3^l in Table 2, h_1 behaves as h_2^l, h_2 behaves as h_1^l and h_3 behaves as h_0^l. This way, separate functions that discriminate locations from the read set, h_0, from locations from the write set, h'_0, assert less bits in its filter reducing the false positive rate, but the filter fails to discriminate locations read from nearby located writes.

- Loc2: In this case, h_3 behaves as h_3^l, h_2 behaves as h_2^l, h_1 behaves as h_1^l and h_0 and h'_0 behaves as h_0^l, i.e. the filter that does not share the hash functions stay the same as in shared 3 configuration, thus discriminating between locations read and written, and the other filters get the locality improvement.

As Figure 11 shows, results for Loc1 scheme are practically the same than those for Loc2 for every benchmark but

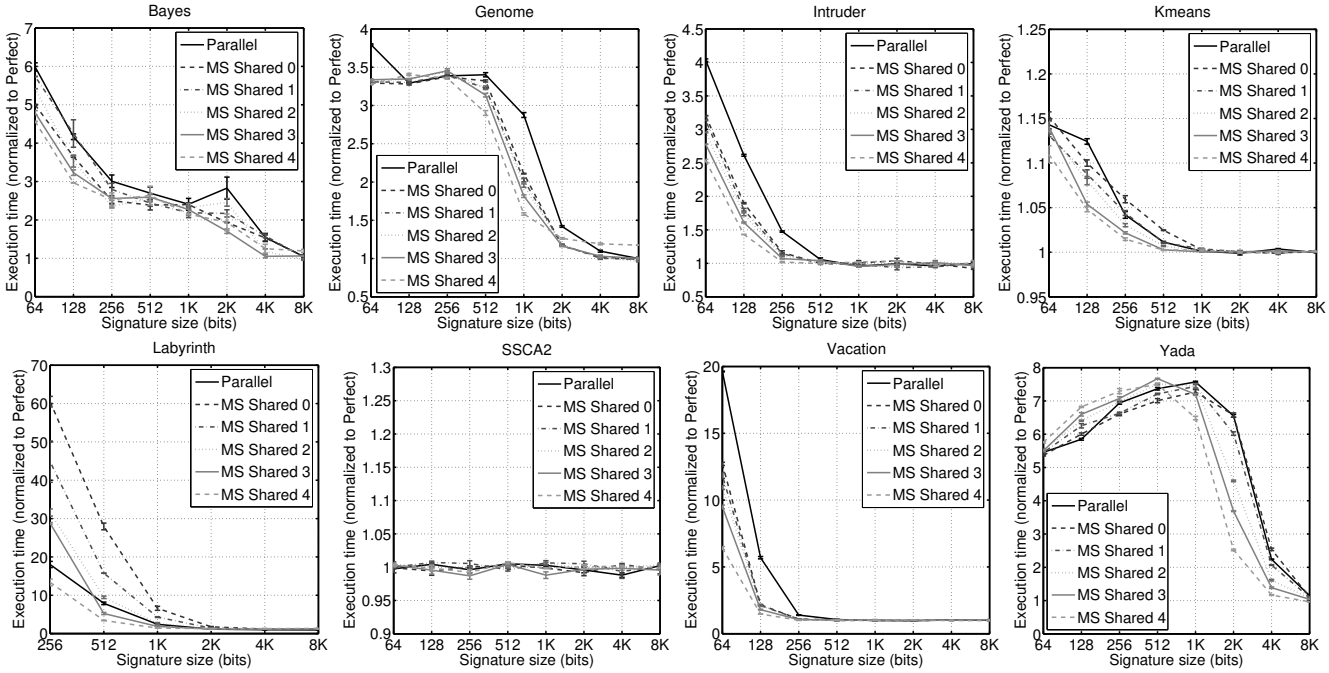


Figure 10: Execution time normalized to perfect signatures (no false positives) comparing parallel signatures to parallel multiset shared signatures varying the shared functions between 0 (no shared functions) and 4 (all functions shared)

for Labyrinth, Genome and Yada. Labyrinth behaves better with Loc2 for small signatures and, Genome and Yada get slightly worse results for small signatures and Loc2. MS shared locality signatures outperform parallel and locality ones in most of cases. Loc2 is 2.4× faster than parallel and 1.5× faster than locality signatures for Bayes and 2Kbit filters. Genome shows a speedup of 2× using Loc2 compared to parallel and 1.6× compared to locality and 1Kbit filters. Intruder shows same speedups from 128bit downwards. Kmeans is 1.1× faster than parallel for small signatures. For Labyrinth, Loc2 shows up to 7× speedup over parallel signatures and 4× over locality for 256bit signatures. Vacation is 3× faster using Loc2 instead of parallel and locality for 64 and 128bit filters. Finally, Yada is 2× faster than parallel and locality using Loc2 for 1Kbit to 4Kbit reaching a peak of 5× over parallel for 2Kbit, though it slows down the execution about 1.2× for small filters.

5.6 Hardware Area and Energy Requirements

In this section an analysis of area and energy requirements for the multiset signatures compared to regular and parallel ones is conducted. Area models from CACTI [29] have been used for that purpose.

Table 3 shows area results 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 regular and parallel signatures and one filter of 8Kbit for multiset ones. CACTI 6.5 [20] was used to model the SRAMs using the 32nm process. Regular signatures have two 4-ported SRAMs (one for the RS and one for the WS) since $k = 4$, with separate read/write ports. Regular multiset signatures have one 8-ported SRAMs and also separate read/write ports. Parallel signatures comprise eight single-ported SRAMs while parallel multiset ones have four double-ported SRAMs. Finally,

Table 2: Example of locality-sensitive hashing: addresses and its indexes for a Bloom with k=4, 2^m=1024

Address	h_0^l	h_1^l	h_2^l	h_3^l
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

parallel multiset shared 3 signatures have three single-ported SRAMs and only one double-ported SRAM. Ports are dual-ended which means that two lines are required per bitline.

A non-linear behavior can be appreciated in area results for the different array sizes. It is related to CACTI's optimization function that searches for the best partition of the cell array depending on time, power and area efficiency. For example, for 4Kbit regular filters 8 byte words were used resulting a 64×64 bit array. Then CACTI found that the best partition was 4 subarrays by splitting bitlines and wordlines in halves. However, for 8Kbit, 8 byte words were used too, resulting a 128×64 bit array. CACTI also found the best partition to be 4 subarrays but it multiplexed the bitlines to share senses amplifiers, which remained constant, thus pro-

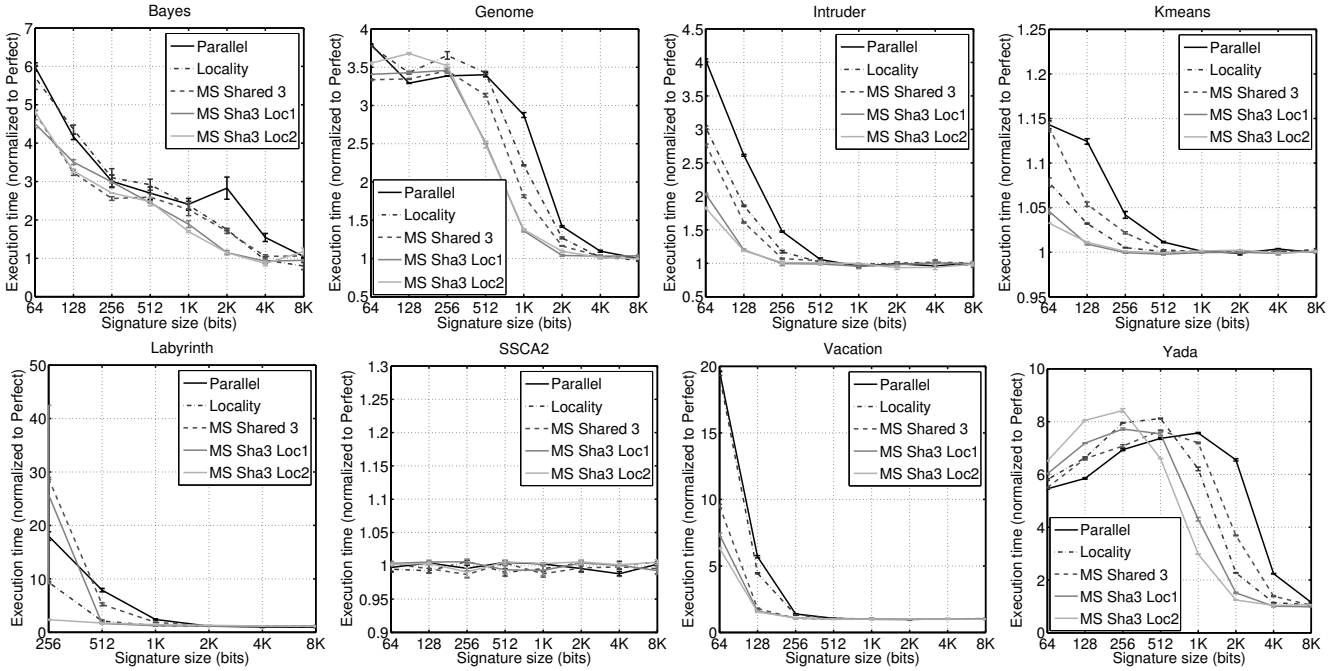


Figure 11: Execution time normalized to perfect signatures (no false positives) comparing parallel signatures, parallel locality signatures and parallel multiset shared 3 signatures enhanced with locality-sensitive hashing (Loc1 and Loc2)

Table 3: Area (mm^2) requirements of regular, parallel and multiset signatures. 32nm technology. $k = 4$ hash functions.

Filter size (2^m)	**4Kbit**	**8Kbit**	**16Kbit**
Regular	0.0707	0.1041	0.2540
Regular MS	0.1860	0.4600	0.6776
Parallel	0.0084	0.0135	0.0292
Parallel MS	0.0191	0.0435	0.0640
Parallel MS 3-shared	0.0098	0.0219	0.0331

Table 4: Dynamic energy per access (nJ). 32nm. $k = 4$

Filter size (2^m)	**4Kbit**	**8Kbit**	**16Kbit**
Regular	0.0074	0.0112	0.0219
Regular MS	0.0180	0.0347	0.0623
Parallel	0.0020	0.0025	0.0047
Parallel MS	0.0030	0.0056	0.0081
Parallel MS 3-shared	0.0026	0.0049	0.0068

viding better area efficiency (memory cell area/total area).

As shown in Table 3, parallel Bloom signatures are significantly better than regular configurations. Actually, they get 8× less area than regular Blooms. On the other hand, regular multiset signatures get the largest area of all. This is because of the multi-ported SRAM which needs 8 ports, four more than the regular one. Regarding the parallel multiset signature, it is about twice larger than the parallel signature due to its double-ported SRAMs. The last configuration of multiset signature, the 3-shared, is the closest to parallel one in terms of area, it is only a 13% larger because of the double-ported SRAM. However, parallel MS shared 3 signatures outperforms parallel ones as seen in Section 5.5. Therefore, the slight increment in area could be worth.

Regarding energy, Table 4 shows a 30% increment in dynamic energy consumption for parallel MS shared 3 signatures.

Finally, Sanchez et al. [25] worked out the hashing logic area for 4 XOR hash functions resulting in one-fifth of the SRAM area. Such an area can be halved using PBX hashing [31] without impact in the performance.

6. CONCLUSIONS

In the context of transactional memory, a multiset signature design is proposed which records both the read and write sets in the same Bloom filter without adding significant hardware complexity. Several designs of multiset signatures are analyzed and evaluated. New problems arise related to hardware complexity and the existence of cross false positives, i.e. new false positives coming from the fact that both sets share the same filter. Additionally, multiset signatures are enhanced using locality-sensitive hashing, proposed by the authors in a previous work.

The proposed multiset and locality-sensitive multiset signatures were implemented in the Wisconsin GEMS simulator, in order to evaluate their performance, and in CACTI in order to evaluate the hardware area and energy requirements. Experimental results show that the multiset approach reduces the false positive rate and improve the execution performance in most of the tested codes, without increasing the required hardware area in a noticeable amount.

We may conclude that multiset and locality-sensitive multiset signatures are a good alternative to non-multiset ones since they yield similar or better performance at without significantly increasing the hardware cost.

7. ACKNOWLEDGMENT

The authors would like to thank Dr. Luke Yen from the

University of Wisconsin, Madison, (now in AMD) for providing his patches to adapt STAMP workloads to GEMS simulator. This work has been supported by the Ministry of Education of Spain with project CICYT TIN2006-01078.

8. REFERENCES

[1] A. R. Alameldeen and D. A. Wood. Variability in architectural simulations of multi-threaded workloads. In *9th Int'l Symp. on High-Performance Computer Architecture (HPCA'03)*, pages 7–18, 2003.

[2] C. Ananian, K. Asanovic, B. Kuszmaul, C. Leiserson, and S. Lie. Unbounded transactional memory. In *11th Int'l. Symp. on High-Performance Computer Architecture (HPCA'05)*, pages 316–327, 2005.

[3] B. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[4] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford Transactional Applications for Multi-Processing. In *IEEE Int'l Symp. on Workload Characterization (IISWC'08)*, pages 35–46, 2008.

[5] L. Carter and M. Wegman. Universal classes of hash functions. *J. Computer and System Sciences*, 18(2):143–154, 1979.

[6] L. Ceze, J. Tuck, P. Montesinos, and J. Torrellas. BulkSC: Bulk enforcement of sequential consistency. In *34th Ann. Int'l. Symp. on Computer Architecture (ISCA'07)*, pages 278–289, 2007.

[7] L. Ceze, J. Tuck, J. Torrellas, and C. Cascaval. Bulk disambiguation of speculative threads in multiprocessors. In *33th Ann. Int'l. Symp. on Computer Architecture (ISCA'06)*, pages 227–238, 2006.

[8] W. Choi and J. Draper. Locality-aware adaptive grain signatures for transactional memories. In *IEEE Int'l. Symp. on Parallel and Distributed Processing (IPDPS'10)*, pages 1–10, 2010.

[9] D. Dice, Y. Lev, M. Moir, and D. Nussbaum. Early experience with a commercial hardware transactional memory implementation. In *14th Int'l. Conf. on Architectural Support for Programming Language and Operating Systems (ASPLOS'09)*, pages 157–168, 2009.

[10] D. Geer. Industry trends: Chip makers turn to multicore processors. *IEEE Computer*, 38(5):11–13, 2005.

[11] L. Hammond, V. Wong, M. Chen, B. Carlstrom, J. Davis, B. Hertzberg, M. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. In *31th Ann. Int'l. Symp. on Computer Architecture (ISCA'04)*, pages 102–113, 2004.

[12] M. Herlihy and J. Moss. Transactional memory: Architectural support for lock-free data structures. In *20th Ann. Int'l. Symp. on Computer Architecture (ISCA'93)*, pages 289–300, 1993.

[13] M. Labrecque, M. Jeffrey, and J. Gregory Steffan. Application-specific signatures for transactional memory in soft processors. In *6th Int'l. Symp. on Applied Reconfigurable Computing (ARC'10)*, 2010.

[14] J. Larus and R. Rajwar. *Transactional Memory*. Morgan & Claypool Pub., 2007.

[15] P. 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.

[16] M. Martin, D. Sorin, B. Beckmann, M. Marty, M. Xu, A. Alameldeen, K. Moore, M. Hill, and D. Wood. Multifacet's general execution-driven multiprocessor simulator GEMS toolset. *ACM SIGARCH Comput. Archit. News*, 33(4):92–99, 2005.

[17] M. Mehrara, J. Hao, P.-C. Hsu, and S. Mahlke. Parallelizing sequential applications on commodity hardware using a low-cost software transactional memory. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI'09)*, pages 166–176, 2009.

[18] 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 *34th Ann. Int'l. Symp. on Computer Architecture (ISCA'07)*, pages 69–80, 2007.

[19] K. Moore, J. Bobba, M. Moravan, M. Hill, and D. Wood. LogTM: Log-based transactional memory. In *12th Int'l. Symp. on High-Performance Computer Architecture (HPCA'06)*, pages 254–265, 2006.

[20] N. Muralimanohar, R. Balasubramonian, and N. Jouppi. CACTI 6.0: A tool to model large caches. Technical Report HPL-2009-85, HP Laboratories, 2009.

[21] R. Quislant, E. Gutierrez, O. Plata, and E. Zapata. Improving signatures by locality exploitation for transactional memory. In *Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT'09)*, pages 303–312, 2009.

[22] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing transactional memory. In *32th Ann. Int'l. Symp. on Computer Architecture (ISCA'05)*, pages 494–505, 2005.

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

[24] B. Saha, A.-R. Adl-Tabatabai, and Q. Jacobson. Architectural support for software transactional memory. In *39st Ann. IEEE/ACM Int'l Symp. on Microarchitecture (MICRO'06)*, pages 185–196, 2006.

[25] D. Sanchez, L. Yen, M. Hill, and K. Sankaralingam. Implementing signatures for transactional memory. In *40th Ann. IEEE/ACM Int'l Symp. on Microarchitecture (MICRO'07)*, pages 123–133, 2007.

[26] A. Shriraman, S. Dwarkadas, and M. Scott. Flexible decoupled transactional memory support. In *35th Ann. Int'l. Symp. on Computer Architecture (ISCA'08)*, pages 139–150, 2008.

[27] A. Shriraman, M. Spear, H. Hossain, V. Marathe, S. Dwarkadas, and M. Scott. An integrated hardware-software approach to flexible transactional memory. In *34th Ann. Int'l. Symp. on Computer Architecture (ISCA'07)*, pages 104–115, 2007.

[28] S. Tomic, C. Perfumo, C. Kulkarni, A. Armejach, A. Cristal, O. Unsal, T. Harris, and M. Valero. EazyHTM: Eager-lazy hardware transactional memory. In *42st Ann. IEEE/ACM Int'l Symp. on Microarchitecture (MICRO'09)*, pages 145–155, 2009.

[29] S. Wilton and N. Jouppi. CACTI: an enhanced cache access and cycle time model. *IEEE Journal of Solid-State Circuits*, 31(5):677 –688, 1996.

[30] L. Yen, J. Bobba, M. Marty, K. Moore, H. Volos, M. Hill, M. Swift, and D. Wood. LogTM-SE: Decoupling hardware transactional memory from caches. In *13th Int'l. Symp. on High-Performance Computer Architecture (HPCA'07)*, pages 261–272, 2007.

[31] L. Yen, S. Draper, and M. Hill. Notary: Hardware techniques to enhance signatures. In *41st Ann. IEEE/ACM Int'l Symp. on Microarchitecture (MICRO'08)*, pages 234–245, 2008.