Improving Signatures by Locality Exploitation for Transactional Memory

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

***Abstract*—Writing multithreaded programs is a fairly complex task that poses a major obstacle to exploit multicore processors. Transactional Memory (TM) emerges as an alternative to the conventional multithreaded programming to ease the writing of concurrent programs. Hardware Transactional Memory (HTM) implements most of the required mechanisms of TM at the core level, e.g. conflict detection. Signatures are designed to support the detection of conflicts amongst concurrent transactions, and are usually implemented as per-thread Bloom filters in HTM. Basically, signatures use fixed hardware to summarize an unbounded amount of read and write memory addresses at the cost of false conflicts (detection of non-existing conflicts).**

In this paper, a novel signature design that exploit locality is proposed to reduce the number of false conflicts. We show how that reduction translates into a performance improvement in the execution of concurrent transactions. Our signatures are based on address mappings of the hash functions that reduce the number of bits inserted in the filter for those addresses nearby located. This is specially favorable for large transactions, that usually exhibit some amount of spatial locality. Furthermore, the implementation do not require extra hardware. Our proposal was experimentally evaluated using the Wisconsin GEMS simulator and all codes from the STAMP benchmark suite. Results show a significant performance improvement in many cases, specially for those codes with long-running, large-data transactions.

***Keywords*-Hardware Transactional Memory; Signatures; Bloom filters; H3 Hashing; Memory locality**

I. Introduction

With the development of multicore processors [1], the common programmer must 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. Shared data in critical sections must be accessed in mutual exclusion to avoid race conditions. Lock-based techniques are used to provide mutual exclusion by serializing the execution of concurrent threads in critical sections. However, narrowing these sections to minimize serialization could lead to convoying, deadlock or priority inversion, problems difficult to detect. In addition, threads and explicit synchronization make constructing abstractions difficult because the programmer must be aware of implementation details. Transactional Memory (TM) [2], [3] 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.

Hardware Transactional Memory (HTM) implements most of the required mechanisms of TM at the core level [4], [5], [6], [7], [8]. HTM 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. So, HTM 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 addresses of such memory reads and writes. Examples of systems that use them are BulkSC [9], LogTM-SE [10], and SigTM [11]. These systems implement signatures as per-thread Bloom filters [12]. Basically, they use fixed hardware to summarize an unbounded amount of read and write memory addresses at the cost of false conflicts (i.e. non-existing conflicts).

Previous signature designs consider all memory addresses as uniformly distributed across the address space. However, in real programs the address stream is not random since it exhibits some amount of spatial and temporal locality. A first contribution of this paper is the proposal of a novel signature design (called, *locality-sensitive signature*) that exploits memory address locality in order to reduce the number of false conflicts. Our proposal defines new address mappings of the hash functions in order to reduce the number of bits inserted in 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 in their read or write sets, but the false conflicts rate remains the same 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 how memory addresses are mapped into the signature filter, its implementation does not require extra hardware. A second contribution of this paper is the implementation of the proposed locality-sensitive

signature in an HTM simulator, in order to show how savings in false conflicts translate into important performance improvements in the execution of concurrent transactions. In particular, we evaluate our proposal using the Wisconsin GEMS [13] simulator and on all codes from the STAMP [14] benchmark suite. Results show a significant performance improvement in many cases, specially for those codes with larger transactions.

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 III we introduce our proposed locality-sensitive signature design, discussing its basics, how they are implemented, and an evaluation compared to the generic signature designs where no memory locality is exploited. Section IV presents the implementation of a locality-sensitive signature on the Winconsin GEMS simulator, and discusses how our signature design may improve the execution performance in several cases, while never harm significantly the performance in any case. Finally, Section V concludes the paper.

II. Background

In the context of TM, each concurrent thread uses its signatures to record all the memory addresses issued when executing a transaction. These addresses 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 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. [15] 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.

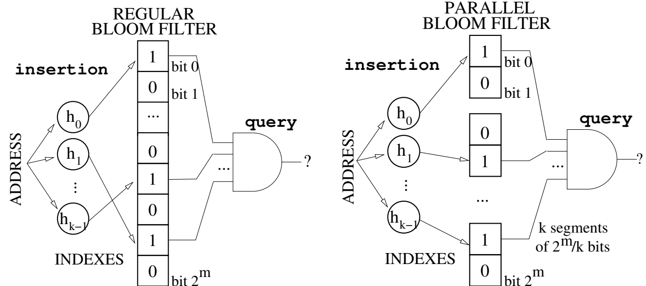

Figure 1. Regular Bloom filter vs. Parallel Bloom filter

Figure 1 shows the implementation of both filters, regular and parallel.

The same paper [15] concludes that Bloom filters should include H3 class hash functions [16], instead of bit-selection hash functions [17], since they are closer to random distribution. However, H3 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 [18].

Notary [18] also proposes a technique to reduce the number of asserted bits in the signature. However, their approach is completely different to ours and it is 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. Our approach, however, require neither software support nor the help of the programmer and, in any case, it could be used combined with every technique in [18].

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 [9]. LogTM-SE [10] uses them in the directory and it ensures paging and context switching with global signatures. SigTM [11] is similar to LogTM-SE. Finally, VTM [8] uses a global signature for cache victimization.

III. Locality Sensitive Signatures

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

For subsequent use, we will consider a Bloom filter that maps a space of 2^n memory addresses, $A = \{0, 1, ...2^n - 1\}$, into an array of 2^m bits (indexes),

$B = \{0, 1, ...2^m - 1\}$, $m \leq n$, through a family of k hash functions, $\{h_0, h_1, ...h_{k-1}\}$. We will consider hash functions of the class H3 because they exhibit a high quality behavior for real memory address streams [15]. Functions in the class H3 basically define a linear transform between an n-bit word and an m-bit word: $h_i : GF(2)^{1\times n} \rightarrow GF(2)^{1\times m}$, being $GF(2)$ the Galois field of two elements [19], under the bitwise xor. Two basic primitives over the Bloom filter are defined: (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. We will denote as $BF(x_0, x_1, ...x_{s-1})$ the set of asserted bits in the Bloom filter after inserting the sequence of s addresses $x_0, x_1, x_2...x_{s-1}$. This set is given by $\bigcup_{i=0}^{s-1} BF(x_i)$, being $BF(x) = \bigcup_{j=0}^{k-1} h_j(x)$.

Note that false positives arise from two situations. First, an address y (not inserted) gives rise to a false positive if there exists x inserted in the Bloom filter and $BF(y) = BF(x)$, $y \neq x$. In such a case, we say that x and y are *aliases*, that is, their mappings through the hash functions h_i are the same. In a Bloom filter, the probability of two addresses being aliases depends on the particular hash functions and the number of them, k. For higher k this probability becomes smaller. Second, a false positive may appear due to the current *occupancy* of the filter. This happens for a non inserted address y after the insertion of s addresses if $BF(y) \subset BF(x_0, x_1, ...x_{s-1})$ and y is not alias of any x_i. However, a false positive takes place. The higher filter occupancy, the higher probability of false positive is expected. 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, a common case [20], occupy a small fraction of the Bloom filter and, hence, show most of the false positives due to aliases and only a few ones due to filter occupancy. However, large data set transactions could exhibit an important amount of false positives due to the high filter occupancy. Note that reducing the number of hash functions, k, helps large data set transactions but not small ones and vice versa [15].

Memory reference locality is a property that may be used to favor small and large transactions at the same time. Our proposal is to build a Bloom filter that maps locations far away each other as normal Bloom filters do, but it maps nearby locations sharing some bits, but not all. This way, k can be chosen highly enough to favor small transactions and, at the same time, favoring large ones by reducing the occupancy of the filter thanks to locality.

Different locality or distance-sensitive hashing schemes have been introduced in the literature. They are used to formulate queries of similarity in metric spaces using compact representations of objects [21], [22], [23]. Inspired by such definitions we will now introduce a formal general signature scheme that takes into account locality of reference to lower the occupancy of the filter when nearby addresses are inserted.

Definition 1: Let be a Bloom filter that maps a space of 2^n memory addresses, A, into a space of 2^m bits, B, $m \leq n$, through a family of k hash functions of the class H3, and let (A, d) and $(\wp(B), d_h)$ be two metric spaces. Such a Bloom filter is called (r, δ)-locality sensitive, with $r \in \mathbb{N}$ and $\delta : \mathbb{N} \rightarrow \mathbb{N}$, if, for any $x, y \in A$, it satifies that,

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

In a Bloom filter designed according to this definition, 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. Regarding to d_h, a usual metric of the distance between two sets is the cardinality of the symmetric difference. Nevertheless, we define $d_h(BF(x), BF(y)) = k - |BF(x) \cap BF(y)|$, that basically measures the number of different hash function outputs when addresses x and y are mapped. As $|BF(x)| = k$, this metric is a half of the cardinality of the symmetric difference of two sets.

Observe in Def. 1 that parameter r is the radius of action of locality-sensitive signatures. Addresses whose distance is greater than r are mapped as though by a generic Bloom filter. As well, the function $\delta(d(x, y))$ can be chosen to increase with $d(x, y)$, in such a way that nearer addresses map into sets of bits less disjoint.

An example of a locality-sensitive signature scheme is shown in Table I, where the output of the k hash functions are computed for a sequence of adjacent locations. Observe that for addresses with $d(x, y) = x \oplus y = 1$, the number of hashing outputs with different values is 1. Addresses with distance 2 are different in no more than 2 hashing outputs. On the other hand, addresses with distance greater than $2^{k-1} - 1$ may have no hashing outputs in common.

A. Implementation

This section introduces an implementation of a locality-sensitive signature scheme by defining some particular hash functions. Our implementation is an instance of Def. 1, where $k = 4$, $d(x, y) = x \oplus y$, $d_h(BF(x), BF(y)) = k - |BF(x) \cap BF(y)|$, $r = 2^{k-1} - 1 = 7$ and:

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

Next, we describe the way to define the H3 matrices corresponding to the parameters above.

As H3 functions under consideration map addresses linearly into indexes, they can be completely characterized by

Table I
EXAMPLE OF LOCALITY-SENSITIVE SIGNATURE: ADDRESSES AND ITS CORRESPONDING H3 INDEXES FOR A BLOOM 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

a matrix in $GF(2)^{n\times m}$ [19]:

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

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 $b = h(a) = [b_{m-1}...b_1 b_0]$ of an n-bit address with binary expression $a = [a_{n-1}...a_1 a_0]$ is computed as follows:

$$[b_{m-1}...b_1 b_0] = [a_{n-1}...a_1 a_0]H. \quad (2)$$

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

$$h(a) = [a_3 a_2 a_1 a_0] \begin{bmatrix} 1 & 0 \\ 1 & 1 \\ 0 & 1 \\ 1 & 0 \end{bmatrix} = [a_3 \oplus a_2 \oplus a_0,\ a_2 \oplus a_1].$$

A Bloom filter with k hash functions is characterized by k H3 matrices $\{H_0, H_1, ..., H_{k-1}\}$.

Hence, the locality-sensitive scheme with the aforementioned parameters can be implemented as k H3 matrices that, when computing the function h_l, $l \in \{0, 1, ..., k-1\}$, the l least significant bits of the address do not participate in the computation. This results in a matrix for the hash function h_l of the form:

$$H_l = \begin{bmatrix} h^l_{n-1,m-1} & h^l_{n-1,m-2} & \cdots & h^l_{n-1,0} \\ h^l_{n-2,m-1} & h^l_{n-2,m-2} & \cdots & h^l_{n-2,0} \\ \vdots & \vdots & & \vdots \\ h^l_{l,m-1} & h^l_{l,m-2} & \cdots & h^l_{l,0} \\ 0 & 0 & \cdots & 0 \\ \vdots & (l \text{ null rows}) & & \vdots \\ 0 & 0 & \cdots & 0 \end{bmatrix}. \quad (3)$$

The example in Table I has been generated following this scheme. This way, the 3 last rows of H_3, the 2 last rows of H_2 and the last row of H_1 are null, whereas H_0 has no null rows.

Since the proposed locality-sensitive signatures can be considered as particular cases of Bloom filters, they do not require any additional hardware. Furthermore, the xor trees could be even simpler insomuch as several hash functions do not make use of certain bits of the address. They can also be implemented directly following a parallel Bloom organization [15]. In addition, this locality-sensitive scheme can be easily combined or extended to other implementations, like the PBX hashing [18].

B. Evaluation

We now present an evaluation of our locality-sensitive signatures compared to the general case, which is discussed in [15]. Consider a sequence of addresses, $x_0, x_1, ...x_{s-1}$, to be inserted in a generic Bloom filter. As each hash function maps one address into one of 2^m possible bits, the probability of one bit being asserted is $\frac{1}{2^m}$, assuming that the outputs of the hash function are uniformly distributed. Hence, the probability of a bit remaining zero is $1 - \frac{1}{2^m}$. After the insertion of s addresses using k hash functions per address, the probability of a bit being zero is

$$p_{\text{ZERO}}(m, k, s) = \left(1 - \frac{1}{2^m}\right)^{sk}, \quad (4)$$

assuming that the outputs of the hash functions are independent.

To get a positive match, all the k bits checked must be asserted. Thus, the probability of a positive is:

$$p_{\text{POSITIVE}}(m, k, s) = (1 - p_{\text{ZERO}})^k = \left(1 - \left(1 - \frac{1}{2^m}\right)^{sk}\right)^k. \quad (5)$$

A test for membership is a true positive for the s addresses inserted so far, but not for the remaining $R - s$ addresses that also get a positive match, being R the number of total positives. So, the probability of getting a false positive is the probability of both getting a positive and not being an inserted address [15]. According to Bayes' rule:

$$p_{\text{FALSE POSITIVE}}(m, k, s) = p_{\text{POSITIVE}}(m, k, s)\frac{R - s}{R} \approx \approx p_{\text{POSITIVE}}(m, k, s). \quad (6)$$

This approximation assumes that the number of total positives in the space under test is much larger than the number of inserted addresses ($R \gg s$).

We now proceed to adapt this general expression to the locality-sensitive signatures of Def. 1. For the sake of simplicity we analyze the case of $d(x,y) = |x-y|$, $\delta(d(x,y)) = 1$ and $r = 1$, that is, contiguous addresses share $k-1$ hash function outputs. Let f be the probability of an address x_j being contiguous to another address x_i already inserted in the filter:

$$f = Pr(d(x_i, x_j) = 1). \quad (7)$$

Now, the probability of filter bits being zero for non contiguous inserted addresses still follows the expression (4). Nevertheless, two contiguous addresses do not assert k bits each one. Instead, they assert only $k+1$ bits between the two of them, according to the definition and the chosen $\delta(d(x,y))$. Since the proportion of contiguous and non contiguous addresses is fs and $(1-f)s$ respectively, the probability of a bit remaining zero for the locality-sensitive scheme can be written as:

$$p^{\text{local}}_{\text{ZERO}}(m,k,s,f) = \left(1 - \frac{1}{2^m}\right)^{s(1-f)k} \left(1 - \frac{1}{2^m}\right)^{sf}. \quad (8)$$

Thus, analogously to expressions (5) and (6), the probability of false positives for the locality-sensitive signature can be expressed as:

$$p^{\text{local}}_{\text{FALSE POSITIVE}}(m,k,s,f) = \left(1 - \left(1 - \frac{1}{2^m}\right)^{s(1-f)k+sf}\right)^k. \quad (9)$$

However, the expression above must be adapted to our locality-sensitive signature with the parameters described in Section III-A. Indeed, we can observe in the example in Table I that not all contiguous addresses share $k-1$ hash function outputs. In fact, this happens only for a half of the contiguous pairs. This involves that the distance $d_h(BF(x), BF(y))$ takes a range of values with certain probability instead of a fixed value for addresses x, y at distance 1. Considering our implementation with $k = 4$, we can infer from Table I the average number of different bits asserted by an address with respect to its preceding contiguous one:

$$\bar{b} = \sum_{i=1}^{k} i \ \Pr(\ d_h(BF(x), BF(y)) = i \mid d(x,y) = 1\ ) = \\ = 1 \cdot 1/2 + 2 \cdot 1/4 + 3 \cdot 1/8 + 4 \cdot 1/8 = 15/8. \quad (10)$$

Finally, this factor is inserted in expression (9) to correct the false positives rate. Therefore, the probability of false

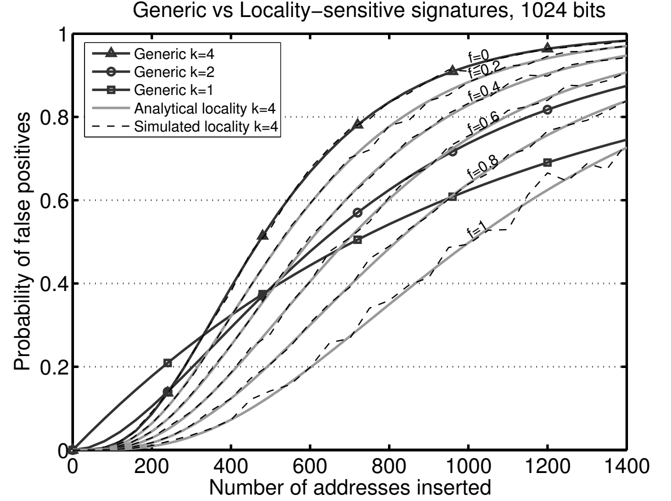

Figure 2. Probability of false positives of generic and locality-sensitive signatures varying f. Simulated values are also shown

positives of our proposed locality-sensitive scheme is:

$$p^{\text{local}}_{\text{FALSE POSITIVE}}(m,k,s,f) = \left(1 - \left(1 - \frac{1}{2^m}\right)^{s(1-f)k+sf\bar{b}}\right)^k. \quad (11)$$

We verified via simulation the expression (11). For that purpose, synthetic address traces were generated fulfilling the definition of f given by expression (7). The probability of false positives was measured using these traces as input to an implementation of our locality-sensitive signatures based on GEMS H3 hash matrices (See Section IV-A). In these simulations the probability of false positives was computed traversing the whole address space keeping track of those non inserted addresses that result in positive matches. Simulation results are depicted in Figure 2 together with the analytical plots derived from expression (11). In addition, three graphs are included corresponding to the generic Bloom filter for different k values, according to expression (6).

Figure 2 shows that best results are obtained with $k = 4$ for small transactions (below 200 inserted addresses). However, for larger transactions the probability of false positives increases rapidly if $k = 4$ and, best results are obtained in this case if $k = 1$. Conversely, our locality-sensitive scheme achieves the best of both situations, as long as the input address sequence exhibits enough locality.

C. Locality in benchmarks

Prior to include 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 to evaluate our proposal (see Section IV-B). Intel's PIN instrumentation tool [24] was used to quickly implement the system. PIN intercepts the

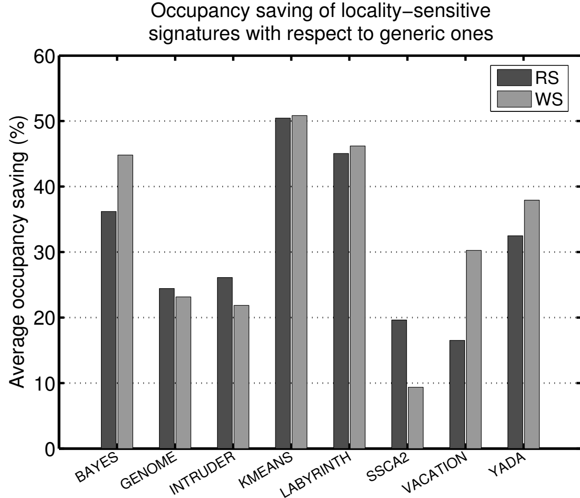

Figure 3. Average Bloom filter occupancy saving of locality-sensitive signature with respect to generic signature

execution of the first instruction of a program and instruments it dynamically. To implement the TM system we developed routines that analyse benchmark instructions until an "open_xact()" is found. Next, instrumentation code keeps track of transaction data sets and the log. In case of a conflict, the requester aborts. When a "commit_xact()" instruction is found, the transaction commits.

Our PIN TM simulator was programmed to record the number of bits set to 1 on each signature (occupancy) of committed transactions, for both generic and locality-sensitive signatures. Figure 3 shows the average occupancy saving percentage of locality-sensitive signatures with respect to generic ones for each benchmark (saving=$\frac{\text{occupancy_generic}-\text{occupancy_locality}}{\text{occupancy_generic}}$). The results have been obtained with 64K-bit filters in order to get rid of occupancy conflicts. Occupancy saving involves to diminish the probability of transactions stalling or aborting due to false positives. Hence, these important savings in the occupancy of the locality-sensitive signature are expected to be translated into execution time improvements.

We can determine an effective value of the parameter f, f_{eff}, for the different benchmarks in order to locate them in Figure 2. Assuming $s \ll 2^m$, that is, the filter is far from saturation, the locality-sensitive filter occupancy can be obtained from the exponent of the expression (11), $s(1-f)k + sf\bar{b}$, from which the values shown in Table II are derived.

IV. Experimental Evaluation

This section describes the simulation environment and methodology (Section IV-A), the STAMP benchmark suite used to evaluate our proposal (Section IV-B) and the experimental results obtained from the simulator (Sections IV-C and IV-D).

Table II
Estimation of effective f for STAMP benchmarks

Benchmark	f_{eff}^{RS}	f_{eff}^{WS}
Bayes	0.85	0.98
Genome	0.48	0.50
Intruder	0.44	0.41
Kmeans	0.96	0.96
Labyrinth	0.86	0.88
SSCA2	0.37	0.18
Vacation	0.32	0.57
Yada	0.67	0.75

A. Simulation environment and methodology

The simulation environment comprises a full system execution-driven simulator called Simics [25] in conjunction with the HTM module GEMS [13] 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 [10] and also includes a detailed timing model for the memory system. Ruby was modified to include our locality-sensitive signature design, both parallel and regular. Same H3 matrices of Ruby have been used to implement the hash functions adding the modifications described in Section III-A.

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 8GB.

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

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 [26].

B. Workloads

All the workloads used in this paper belong to the Stanford's STAMP suite [14]. This suite is designed for

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

Table III
WORKLOADS: INPUT PARAMETERS AND TM CHARACTERISTICS

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

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 (PLT) (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 to avoid inserting in the signature those reads. Regarding change (ii), a modified version of the memory pool library provided by STAMP was used.

Table III 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. A metric for locality in benchmarks, column "Xact locality", was obtained from Table II. The last columns stand for the average and the maximum values of RS and WS size distributions in cache lines.

C. GEMS results

The results obtained from each workload simulation are shown in Figures 4 and 5. Figure 4 depicts the execution time, measured in Ruby cycles, normalized to perfect signatures (no false positives) comparing generic and locality-sensitives schemes for both regular and parallel implementations. Figure 5 shows the results in terms of speedup. Observe that locality-sensitive signatures perform better or similar than generic ones for most cases. They slightly increase the execution time only for a few configurations.

From these results we distinguish three different groups of behavior, namely:

1) **SSCA2:** This workload exhibits the smallest transactions of the whole suite. RS and WS maximum sizes are only 3 and 2 cache lines, respectively. Moreover, the benchmark spends most of the time outside transactions (see Table III). Hence, SSCA2 is not signature size sensitive, as Figures 4 and 5 show.
2) **Kmeans, Vacation, Intruder and Labyrinth:** These benchmarks show similar behavior when signature size decreases. Locality-sensitive signatures reduce, in some cases considerably, the execution time. They always either outperform or match the performance of generic signatures.
   Kmeans is low contended and spends only 6% of time in transactions, so it is not too signature dependent (Figure 5 shows a speedup of only 1.16 for the best case). Even so, parallel locality-sensitive signatures reduce execution time of generic ones when 128-bit filters are used since transactions are medium size and exhibit high locality (see Table II).
   In Vacation, locality-sensitive signatures match the execution time of generic signatures because of mid-locality and medium-to-small transactions. It is 25% better for 128-bit signatures since 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 for 64-bit signatures and 4 hash functions.

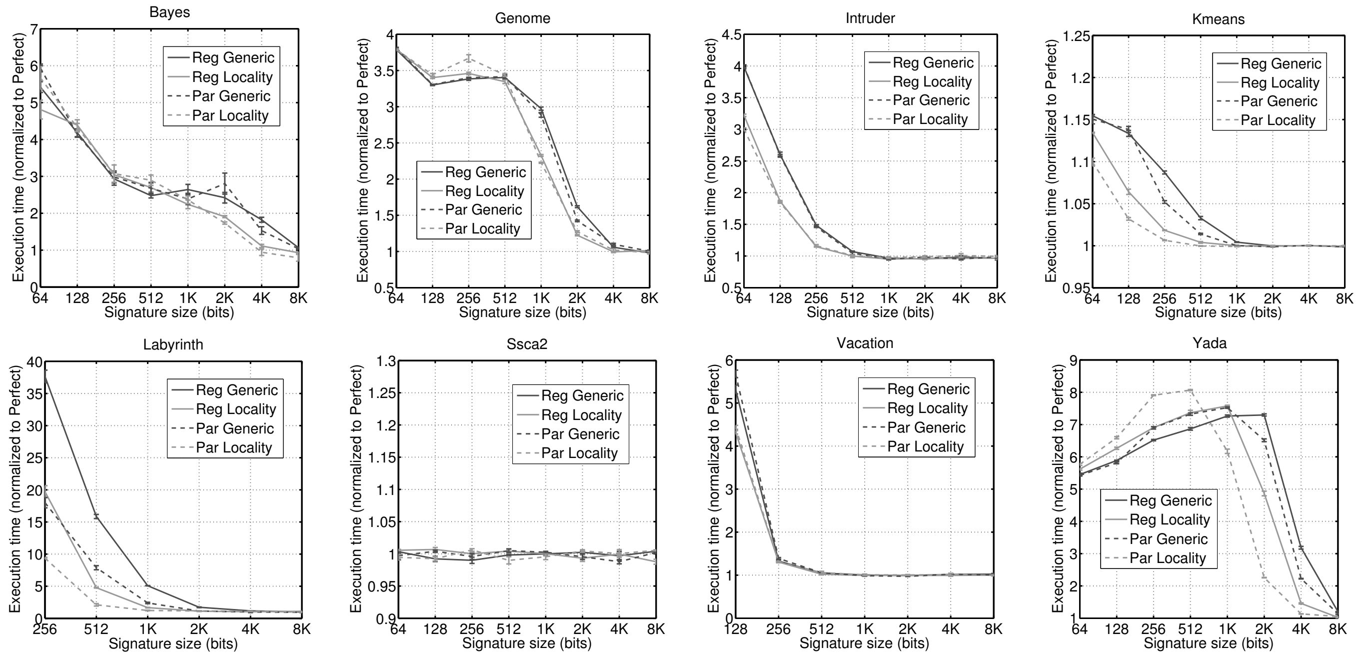

Figure 4. Execution time normalized to perfect signature (no false positives) comparing generic and locality-sensitives schemes for both regular and parallel implementations, with k=4

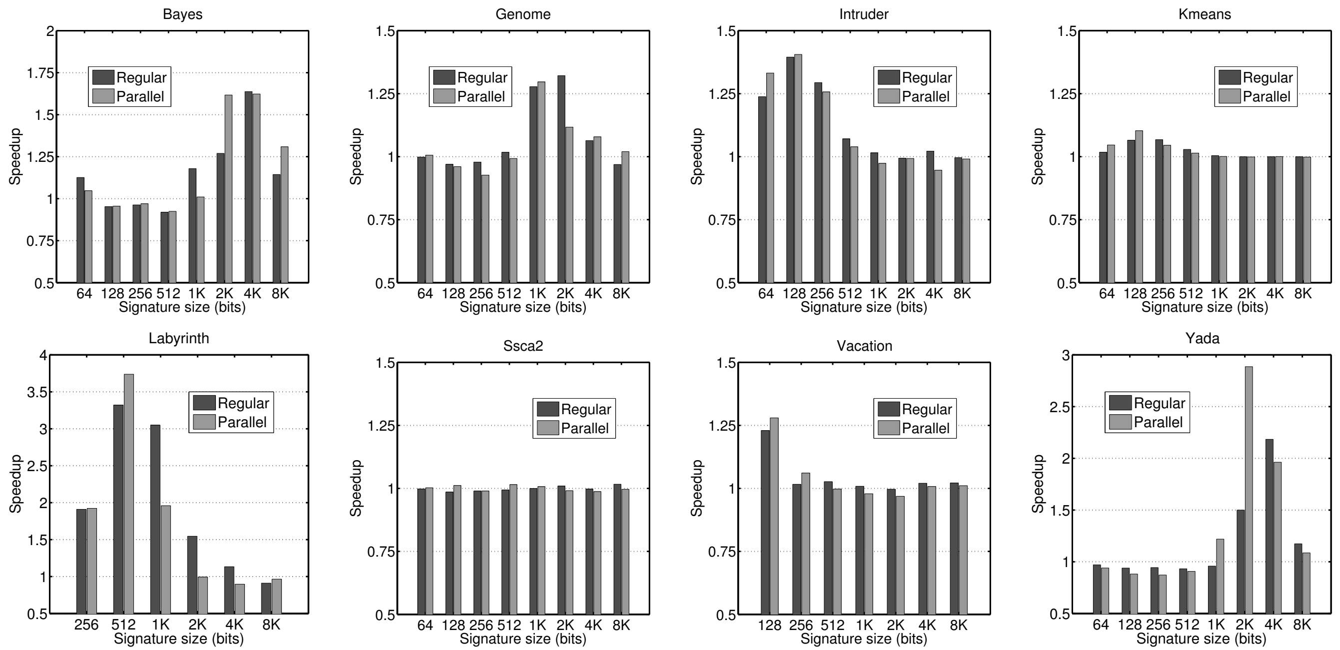

Figure 5. Speedup of locality-sensitive signatures over generic versions for both regular and parallel implementations, with k=4

Table IV
RETRIES PER TRANSACTION AND AVERAGE ABORT DELAY IN CYCLES (RETRIES/ABORT_TIME) FOR BAYES, YADA AND GENOME BENCHMARKS. SIGNATURE SIZE IS 256 BITS.

Signature	Bayes	Genome	Yada
Reg Generic	41.5/2467	0.139/46603	5.2/2844
Reg Locality	33.7/2495	0.137/47538	4.6/3512
Par Generic	46.3/2324	0.139/46503	5.2/2844
Par Locality	44.3/2385	0.136/51221	4.5/4216

Intruder shows a behavior similar to Vacation with 25% execution time reduction for small signatures.
Finally, Labyrinth shows up a great improvement in performance. Regular locality-sensitive signatures exhibit a speedup of almost 4 for 512-bit signatures. High locality, large transactions in average (both RS and WS) and high contention make locality-sensitive signatures to remarkably outperform generic ones.

3) **Bayes, Genome, Yada:** Locality-sensitive signatures yield better results than generic ones for large signatures but they are slightly worse for small ones in these benchmarks. This behavior is related to the way 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.
Table IV shows the number of retries (aborts) per transaction along with the average abort delay for the three benchmarks and the four different signatures when the signature size is 256 bits. These statistics show up that locality-sensitive signatures decreases the number of aborts but the abort delay increases. Hence, transactions are able to run for a longer time before encountering a conflict, since locality-sensitive signatures yield better false conflict rate even on small signatures, but on abort they must undo the log that is now longer than if the conflict was detected earlier. Note that decreasing false positive rate in signatures does not necessarily involve a direct improvement in performance. Other factors, like abort patterns, may prevail.
Even despite of the aforementioned execution time loss, locality-sensitive signatures still perform better than generic ones for large signatures in these benchmarks. Yada shows a significant execution time saving for 2K-bit signatures. Regular locality-sensitive signatures are up to 3 times faster than generic ones.

D. Saving hardware

Locality-sensitive signatures can be thought of to enable smaller signature sizes opposed to only improving false-conflicts.

Figure 4 shows that Yada and Labyrinth yield the same results if we use a parallel locality-sensitive signature with half size of generic ones. Intruder and Genome behave similar but we get slightly worse results halving the signature. Vacation and Bayes could be in the same group as Yada and Labyrinth, however, we should not halve the signature size from 256 bit downward.

V. CONCLUSIONS

A novel signature design that takes advantage of locality of memory references has been proposed in the context of transactional memory. Our proposal, called locality-sensitive signature, is based on Bloom filters with H3 hash functions, and it is aimed to reduce the number of false positives existing in generic schemes with the expectation of improving the execution of concurrent transactions.

We have implemented locality-sensitive signatures in the Wisconsin GEMS simulator and we have evaluated them using the STAMP benchmark suite. Results show that our signature proposal improves benchmark performance in most cases, specially for large-set long-running transactions, where the probability of false positives may be higher. As a remarkable feature our proposal do not require extra hardware with respect to generic signatures.

We may conclude that parallel locality-sensitive signatures are a good alternative to generic ones since they yield similar or better performance at the same cost.

ACKNOWLEDGMENT

The authors would like to thank Dr. Luke Yen from the University of Wisconsin, Madison, 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.

REFERENCES

[1] D. Geer, “Industry trends: Chip makers turn to multicore processors,” *IEEE Computer*, vol. 38, no. 5, pp. 11–13, 2005.

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

[3] 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)*, 1993, pp. 289–300.

[4] 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)*, 2004, pp. 102–113.

[5] 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)*, 2005, pp. 316–327.

[6] 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)*, 2006, pp. 254–265.

[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)*, 2006, pp. 227–238.

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

[9] 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)*, 2007, pp. 278–289.

[10] 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)*, 2007, pp. 261–272.

[11] 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)*, 2007, pp. 69–80.

[12] B. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[13] 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*, vol. 33, no. 4, pp. 92–99, 2005.

[14] 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)*, 2008, pp. 35–46.

[15] 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)*, 2007, pp. 123–133.

[16] L. Carter and M. Wegman, "Universal classes of hash functions," *J. Computer and System Sciences*, vol. 18, no. 2, pp. 143–154, 1979.

[17] M. V. Ramakrishna, E. Fu, and E. Bahcekapili, "Efficient hardware hashing functions for high performance computers," *IEEE Trans. on Computers*, vol. 46, no. 12, pp. 1378–1381, 1997.

[18] 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)*, 2008, pp. 234–245.

[19] H. Vandierendonck and K. De Bosschere, "XOR-based hash functions," *IEEE Trans. on Computers*, vol. 54, no. 7, pp. 800–812, 2005.

[20] J. Chung, H. Chafi, C. Cao Minh, A. McDonald, B. D. Carlstrom, C. Kozyrakis, and K. Olukotun, "The common case transactional behavior of multithreaded programs," in *12th Int'l Symp. on High Performance Computer Architecture (HPCA'06)*, 2006, pp. 266–277.

[21] M. Charikar, "Similarity estimation techniques from rounding algorithms," in *34th Ann. ACM Symp. on Theory of Computing (STOC'02)*, 2002, pp. 380–388.

[22] P. Indyk and R. Motwani, "Approximate nearest neighbors: towards removing the curse of dimensionality," in *30th Ann. ACM Symp. on Theory of Computing (STOC'98)*, 1998, pp. 604–613.

[23] A. Kirsch and M. Mitzenmacher, "Distance-sensitive bloom filters," in *8th Workshop on Algorithm Engineering and Experiments (ALENEX'06)*, 2006, pp. 41–50.

[24] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Reddi, and K. Hazelwood, "PIN: Building customized program analysis tools with dynamic instrumentation," in *ACM SIGPLAN Conf. on Programming Languages Design and Implementation (PLDI'05)*, 2005, pp. 190–200.

[25] 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*, vol. 35, no. 2, pp. 50–58, 2002.

[26] 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)*, 2003, pp. 7–18.