LS-Sig: Locality-Sensitive 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) is an alternative to conventional multithreaded programming to ease the writing of concurrent programs. In the context of unbounded TM, concurrent threads may use hardware signatures to record all the memory addresses issued inside a transaction to detect conflicts. Signatures are usually implemented as per-thread fixed hardware Bloom filters that summarize a very large amount of read and write memory addresses at the cost of false conflicts (detection of non-existing conflicts).**

In this paper, to reduce the probability of false conflicts, a novel signature design that exploits spatial locality is proposed. The design is based on new hash function mappings, so that nearby located addresses share some bits inserted in the filters. This is favorable particularly for large transactions that usually exhibit some amount of spatial locality. Besides, its implementation does not require extra hardware. The proposed signature was experimentally evaluated using the GEMS simulator and all the codes of the STAMP benchmark suite. In most cases, the results show significant improvement, particularly in the codes that involve long-running, large-data transactions.

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

I. Introduction

Writing multithreaded parallel programs is a complex task. It forms a major obstacle in utilizing multicore processors. Parallelism introduces non-determinism that must be controlled by careful designing 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 (ensure consistency). Lock-based techniques are used traditionally to provide mutual exclusion by serializing the execution of critical sections from concurrent threads. By narrowing these sections, the thread serialization can be minimized, but at the cost of increasing the lock overhead and the risk of deadlock. In general, there is a trade-off between serialization and lock contention, depending on the lock granularity. In addition, locks have other disadvantages, like convoying or priority inversion, and difficulties in problem detection and solving. Finally, lock-based techniques lack effective mechanisms of abstraction and composition, because explicit synchronization requires that the programmer is aware of implementation details.

Transactional Memory (TM) [1], [2] is emerging as an alternative to the conventional multithreaded programming to ease the writing of concurrent programs. TM introduces the concept of transaction as a convenient abstraction for coordinating concurrent access to shared data, allowing semantics to be separated from implementation. A transaction is a block of computations that appear to be executed with atomicity and isolation. Thus, transactions replace a pessimistic lock-based model by an optimistic model and they solve abstraction and composition problems.

TM systems execute transactions in parallel, committing the non-conflicting ones. A conflict occurs when a memory location is concurrently accessed by several transactions and at least one access is a write. In such a case, the conflict must be resolved to preserve atomicity. Conflicts can be identified and solved while the transactions are running (eager conflict detection) or at commit time (lazy conflict detection).

This paper focuses on hardware implementation of TM, the systems that map basic TM operations on hardware. Earlier systems exploited the fact that conflicting requests to transactional memory locations can be observed through cache coherence mechanism [3], [4], [5], [6], [7]. These implementations can support only transactions bounded in time and size. More recently, the TM proposals include hardware support for very large transactions by adding some mechanisms to handle transactions that overflow the cache hierarchy [3], [8], [9], [10], [11], [12], [13], [14], [15]. Other approaches, like hybrid TM models, simplify the hardware support by not implementing some complex transactional features [16], [17], [18]. Finally, there are certain systems that provide hardware support to speed up parts of software TM implementation (STM) [19], [20], [21].

All the above mentioned TM systems must record all memory reads and writes during the execution of transactions for detecting and resolving conflicts. Recently, signatures are proposed as hardware devices to store the addresses of such memory access. Systems that use these signatures include BulkSC [22], LogTM-SE [11], SigTM [23], FlexTM [21], FASTM [24], DynTM [25] and STMlite [26], the last one being a full software implementation. These systems implement signatures as per-thread Bloom filters [27], one for recording read addresses and the other for recording write addresses. Basically, they use fixed hardware to summarize a very large amount of read and write memory addresses at the cost of false conflicts (i.e. non-existing conflicts).

Previous signature designs consider that all memory addresses are uniformly distributed across the address space. However, in real programs the address stream is not random as it exhibits some amount of locality. The first and main contribution of this paper is proposing a novel signature design based on Bloom filters, called *locality-sensitive signature*

(LS-Sig), which exploits memory reference locality to reduce the probability of false conflicts. The proposal defines new address mappings of the hash functions to reduce the number of bits inserted in the filter (occupancy) for the addresses with spatial locality. That is, nearby memory locations share some bits of the Bloom filter. As a result, false conflicts are significantly reduced in transactions that exhibit spatial locality in their read or write sets, but the false conflict rate remains unalterable for transactions that do not exhibit locality at all. This is favorable particularly for large transactions that usually present a significant amount of spatial locality. In addition, as the proposal is based on new locality-aware hash mappings, its implementation does not require extra hardware.

The second contribution of this paper is implementing the proposed LS-Sig in a hardware TM simulator to show how savings in false conflicts translate into important performance improvements while executing concurrent transactions. In particular, different variants (hash mappings) of LS-Sig were evaluated on codes from the STAMP [28] benchmark suite using the Wisconsin GEMS [29] simulator. Also, these signatures were compared with other state-of-the-art implementations available in literature. In most cases, the results show significant improvement in performance, particularly for codes with larger transactions.

The remainder of the paper is organized as follows. In the next section, a brief review of signatures is presented, together with a description of designs and implementation in related works. In Section III, the proposed LS-Sig design is introduced with a discussion of its basics, followed by its comparison with the generic signature designs for probabilistic evaluation. Section IV deals with the implementation of different variants of LS-Sig on the Winconsin GEMS simulator, and discusses how LS-Sig can improve the execution performance in several cases. Finally, Section V sums up the conclusions of this study.

II. Background

In the context of HTM, concurrent threads use hardware signatures to record all the memory addresses issued within a transaction. These addresses are sorted out into a read set (RS) and a write set (WS). Each thread uses a separate private signature for storing each set. As a conflict detection device, signatures should not tolerate false negatives (undetected true conflicts), but may assume false positives (false conflicts). In addition, as the sizes of RS and WS are not known in advance, signatures should not limit the number of addresses to be tracked. Finally, test and insertion operations should be fast.

Fulfilling the above requirements, Ceze et al. [4] propose 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. A 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 if those k bits are asserted.

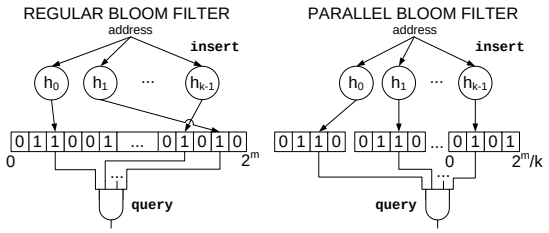

Fig. 1. Regular Bloom filter vs. Parallel Bloom filter designs.

Bloom filters are also known as true or regular Bloom filters. Sanchez et al. [30] propose parallel Bloom filter as an alternative hardware-efficient implementation to regular Bloom filters. The regular filter is implemented as a k-ported SRAM, and the parallel one as k 1-ported SRAMs, yielding the same or better false positive rate. Fig. 1 shows the implementation of both regular and parallel filters. Sanchez et al. conclude that Bloom filters should use H3 class hash functions [31], instead of bit-selection ones [32], as they are closer to random distribution. However, H3 hashing is hardware-expensive and needs an XOR tree per hash bit.

An alternative hardware-efficient implementation of hash functions, Page-Block-XOR hashing (PBX), has been proposed in [33]. The authors of this proposal use the concept of entropy to find the most random bits of the address, which reduce the hardware complexity of the hash functions. Notary [33] also proposes a technique to reduce the number of asserted bits in the signature. However, his approach is different as it is based on segregating addresses into private and shared sets, and then recording only the shared addresses. This solution requires support at the compiler, runtime/library and operating system levels. In addition, the programmer must define which objects are private and which are shared. The approach proposed here, however, requires neither software support nor programmer's help. In fact, it could be used in combination with every technique in [33].

More recently, Choi and Draper [34] proposed adaptive grain signatures that store the history of transaction aborts and dynamically change the input bit range to the hash functions on the abort history. The aim of this design is to reduce the number of false positives that affect execution performance.

Several hardware supported TM systems adopt signatures rather than keeping transactional state in cache hierarchy. Modifying caches to track transactional information has been proved to be beset with major constraints, because 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 adding additional hardware. Signatures are used to enforce sequential consistency in BulkSC [22]. LogTM-SE [11] uses them in the directory and ensures paging and context switching with global signatures. SigTM [23] is similar to LogTM-SE. VTM [10] uses a global signature for cache victimization. Finally, signatures are also proposed as a good solution to support conflict detection in hardware TM implementations on FPGA-based computing systems [35].

III. Locality Sensitive Signatures

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

It is clarified that what will be considered in the ensuing discussion here is a Bloom filter 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, $\{h_0, h_1, ..., h_{k-1}\}$. Hash functions of class H3 will be used, because they exhibit high quality behavior for memory address streams [30]. Functions in class H3 basically define a linear transform between an n-bit word and an m-bit word: $h_i : GF(2)^{1\times n} \rightarrow GF(2)^{1\times m}$, $GF(2)$ being the Galois field of two elements [36], under the bitwise XOR. Two basic operations are defined over the Bloom filter: (i) inserting an address x by asserting its mapped bits ($h_i(x) = 1$), and (ii) checking if an address has been already inserted by testing if all its corresponding mapped bits are set to 1. Let $BF(x_0, x_1, ..., x_{s-1})$ be 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 $BF(x) = \bigcup_{j=0}^{k-1} h_j(x)$.

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

In general, small data set transactions, a common case [37], occupy a small fraction of the Bloom filter and, hence, show false positives most of which are due to aliases and only a few due to filter occupancy. However, large data set transactions show many false positives, which are due to high filter occupancy. Reducing the number of hash functions, k, helps large data set transactions, but not the small ones [30].

Memory reference locality is a property that may be used to favor small and large transactions simultaneously. This paper proposes to build a Bloom filter that maps locations far away from each other as normal Bloom filters do, while the nearby locations are mapped sharing some bits. This way, one can choose k to be high enough to favor small transactions, and, at the same time, the large ones too by reducing the occupancy of the filter, thanks to locality.

Different locality- or distance-sensitive hashing schemes are available in the literature. They are used to formulate queries of similarity in metric spaces using compact representations of objects [38], [39], [40]. Motivated by these definitions, a formal general signature scheme is introduced here that can take into account the locality of reference to reduce the occupancy of the filter when nearby addresses are inserted.

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

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

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 ((r, δ)-LS), with $r \in \mathbb{N}$ and $\delta : \mathbb{N} \rightarrow \mathbb{N}$, if, for any $x, y \in A$, it satisfies that,

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

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

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

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

A. Implementation

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

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

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

Next, a set of H3 matrices are proposed according to the parameters described above. As H3 functions under consideration map addresses linearly into indexes, they can be completely characterized by a matrix in $GF(2)^{n \times m}$ [36]:

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

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 that computes the bit j of the index. The hash output $b = h(a) = [b_{m-1}...b_1b_0]$ of an n-bit address with binary expression $a = [a_{n-1}...a_1a_0]$ corresponds to a mapping $GF(2)^{1\times n} \rightarrow GF(2)^{1\times m}$, which is computed as follows:

$$[b_{m-1}...b_1b_0] = [a_{n-1}...a_1a_0]H. \quad (3)$$

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

$$h(a) = [a_3a_2a_1a_0] \begin{bmatrix} 1 & 0 \\ 1 & 1 \\ 0 & 1 \\ 1 & 0 \end{bmatrix} = [a_3 \oplus a_2 \oplus a_0,\ a_2 \oplus a_1].$$

A general Bloom filter with k hash functions is characterized by k H3 matrices $\{H_0, H_1, ..., H_{k-1}\}$. To obtain an LS scheme that is compatible with δ function in expression 1, each of these functions is transformed by masking each H_l with the following square matrix, $M_l \in GF(2)^{n\times n}$, obtained by nullifying the last l elements of the identity matrix:

$$M_l = diag(\overbrace{1, 1, \cdots, 1}^{n-l}, \overbrace{0, 0, \cdots, 0}^{l}) = \quad (4)$$
$$= \begin{bmatrix} 1 & & & \cdots & \\ & 1 & & \cdots & \\ & & 1 & \cdots & \\ \underbrace{\qquad}_{n-l \text{ ones}} & & & \overbrace{\qquad}^{l \text{ zeros}} & \\ & \cdots & & 0 & \\ & \cdots & & & 0 \end{bmatrix}.$$

The proposed LS Bloom is now characterized by k H3 matrices $\{LH_0, LH_1, ..., LH_{k-1}\}$, where the last l rows in each one have been nullified, that is:

$$LH_l = M_l \cdot H_l. \quad (5)$$

Hereafter, this signature scheme will be referred to as LS-Sig.

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

B. Features

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

A hash function is expected to be uniform, that is, inputs should be equitably mapped over their output range. In this way, the number of inputs colliding on the same element of the destination space should be the same, avoiding aliases to concentrate on certain elements. In the case of a linear transformation, as the one in expression 3, the alias distribution is directly related to the null space associated with the linear function. The null space is given by those vectors x such that $xH = 0$, 0 being the null vector of the destination space. It is commonly denoted as $N(H)$. If two inputs a, b are considered aliases, then $aH = bH \Rightarrow (a - b)H = 0$ and therefore, $a - b \in N(H)$. All linear combinations of vectors in the null space are aliases of 0, and an alias of a given input can be generated by adding a vector of the null space to it.

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

Focusing on the computation of locality hash matrices according to expression 5, it can be seen that the dimension of the null space, and consequently the uniformity of the hash function, will not be affected if the rows being nullified are linearly dependent on the remaining ones. This condition is easy to fulfill if the input space is larger than the destination space, as has already been assumed ($m \leq n$), and the number of nullified rows does not exceed the aforementioned maximum kernel dimension, that is, $k \leq n - m$.

This situation was verified for all the matrices used in the experiments of this work. For example, the effect of nullifying the last rows on one of these matrices is shown in Fig. 2, which depicts the hash matrix and its associated null space. Dots represent the non-zero matrix coefficients. For the null spaces, a vector basis is shown in the reduced row echelon form, which makes their visual comparison easy. It is to be noted that when nullifying the l-th last row of the hash matrix, vectors with only the l-th bit asserted appear in the the null space basis. Nevertheless, it is important that the null space dimension remains at its maximum desirable value $n - m$ for all LS matrices.

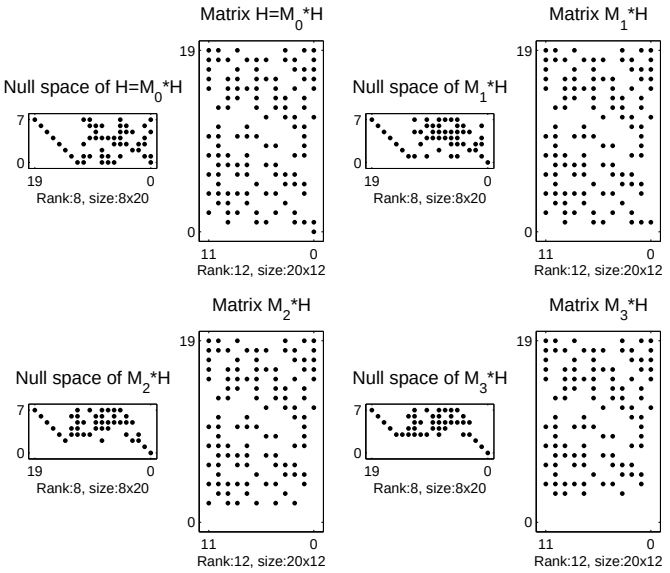

Fig. 2. Effect of nullification of rows of a hash matrix on its null space ($N(H) \cdot H$ is null matrix).

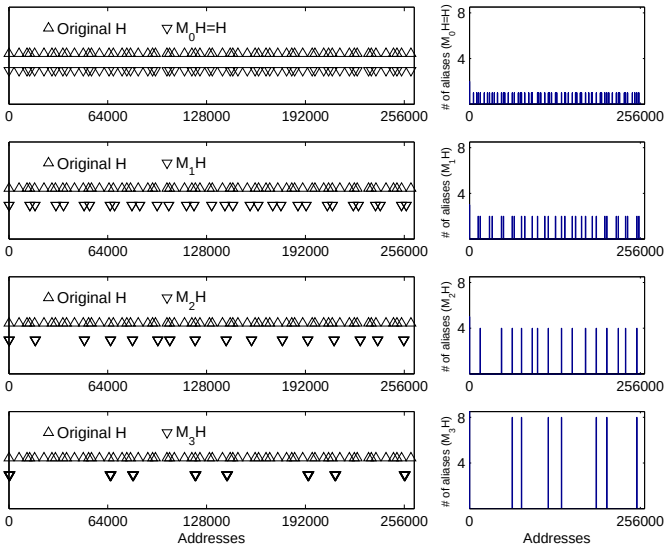

Fig. 3. Pattern of the aliases of zero for LS hashing matrices in Fig 2 (left), and its density distribution (right).

What is happening here is that the pattern of aliases is changing as rows are nullified, but the number of aliases remains unchanged. This fact can be observed in Fig. 3 which shows the aliases of zero for matrices of Fig. 2. As the number of nullified rows increases, aliases concentrate in batches of consecutive addresses (nullifying l rows involves batches of size 2^l). This can be seen in the alias density histogram of the figure. Therefore, the way the locality sensitive hash matrices are generated does not alter the null space dimension, and consequently the number of aliases will be the same for the starting hash functions.

As regards implementation cost, no additional hardware is required, because the proposed LS-Sig can be considered a special case of Bloom filter. Besides, the XOR trees could be even simpler, because several hash functions do not make use of certain bits of the address. LS-Sig can also be implemented directly following a parallel Bloom organization [30]. In addition, this locality-sensitive scheme can be easily combined with or extended to other implementations, like the PBX hashing [33], as shown in Section IV-E.

C. Evaluation

In this section, the proposed LS signature is evaluated in relation to the general case discussed in [30]. For this, a sequence of addresses, $x_0, x_1, ...x_{s-1}$, is considered for insertion into 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 functions 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

$$p_{\text{ZERO}}(m,k,s) = \left(1 - \frac{1}{2^m}\right)^{sk}, \tag{6}$$

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:

$$p_{\text{POSITIVE}}(m,k,s) = (1 - p_{\text{ZERO}})^k = \left(1 - \left(1 - \frac{1}{2^m}\right)^{sk}\right)^k. \tag{7}$$

A test for membership is a true positive for the s addresses inserted thus far, but not for the remaining $R - s$ addresses, which also get a positive match, R being the number of total positives. So, the probability of getting a false positive is the probability of getting a positive and also of not being an inserted address [30]. According to Bayes' rule

$$p_{\text{FALSE POSITIVE}}(m,k,s) = p_{\text{POSITIVE}}(m,k,s)\frac{R-s}{R} \approx \approx p_{\text{POSITIVE}}(m,k,s). \tag{8}$$

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

Next, a general expression of the false positive rate is obtained for the locality-sensitive signatures of Def. 1 with δ function of expression 1 and the rest of conditions in Section III-A. The probability of false positives is expected to depend on the spatial proximity between addresses. For instance, a new address y is considered for insertion in the Bloom filter, x being the nearest address already inserted. Considering the stochastic variable t, assume that f_t is the probability that δ function is equal to t, for any address to be inserted with its nearest address already in the filter,

$$f_t = Pr(\delta(d(y, \min_{x \text{ inserted}} d(x,y))) = t), 1 \le t \le k. \tag{9}$$

Here, $\min_x f(x)$ denotes the value of x where a given function $f(x)$ is minimum. This definition excludes the repetitions of addresses during insertion; hence, f_0 has not been taken into

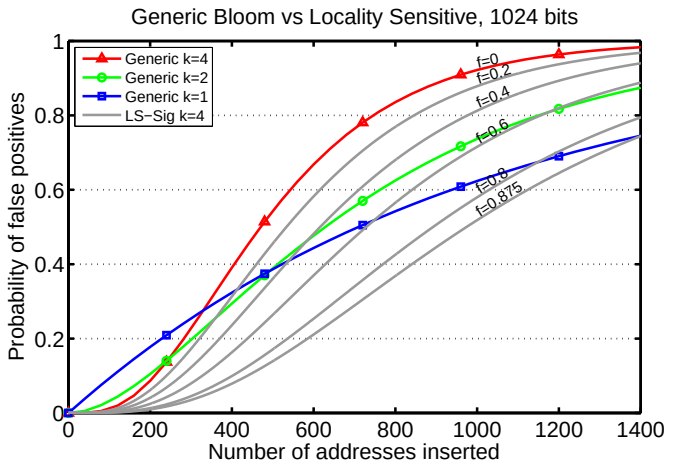

Fig. 4. Probability of false positives of generic and LS signatures varying the parameter $f = \sum_{t=1}^{3} f_t$ (the higher the f, the more the locality).

account and consequently t goes from 1 to k, as δ definition saturates beyond k.

This way, the probability of filter bits being zero for distant inserted addresses continues to follow the expression 6. Nevertheless, neither of the two near addresses x, y (having distance less than 2^{k-1}) will assert k bits. Instead, they assert only $k+\delta(d(x,y))$ bits in total. Making use of the probabilities of expression 9, the probability of a bit remaining zero for the locality-sensitive scheme can be written thus:

$$p^{\text{local}}_{\text{ZERO}}(m,k,s) = \left(1 - \frac{1}{2^m}\right)^{s\sum_{t=1}^{k} t\cdot f_t} . \qquad (10)$$

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

$$p^{\text{local}}_{\text{FALSE POSITIVE}}(m,k,s) = \left(1 - \left(1 - \frac{1}{2^m}\right)^{s\sum_{t=1}^{k} t\cdot f_t}\right)^{k} . \qquad (11)$$

Fig. 4 shows the analytical evaluation of false positive probability for the original Bloom filter (expression 8) for several k values, and the proposed LS scheme (expression 11) for $k = 4$. To parameterize the evaluation, $f = \sum_{t=1}^{k-1} f_t$ was introduced as the probability of an address being near to some inserted address. Consequently, $1 - f = f_k$ is the probability of being far from those already in the filter. As the parameter f gathers probabilities for different distances, a Zipf distribution was chosen, which is often assumed for modeling reference locality [41], [42]. In this way, expression 11 was evaluated with $f_2 = \frac{1}{2}f_1$, $f_3 = \frac{1}{3}f_1$. It is to be noted that expressions 10 and 11 are valid only for the distance metric under consideration. Likewise, this metric introduces some constraints to possible f_t values. Considering a monotonically increasing consecutive sequence of different addresses, it was thus fulfilled that $f_4 \geq \frac{1}{8}$, and $f_1 \leq \frac{1}{2}$. The lower bound of f in the plot was derived from these constraints.

Whereas low values of k are advantageous for large transactions, and high values of k for small ones, with a generic Bloom filter, it can be inferred from Fig. 4 that the LS scheme can achieve the benefits of both situations if the address sequence exhibits medium/high spatial locality.

TABLE II
PROBABILITIES DEFINED BY EXP. 9 FOR STAMP CODES (THE HIGHER THE f_1 THE MORE THE SPATIAL LOCALITY IN TRANSACTIONS).

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

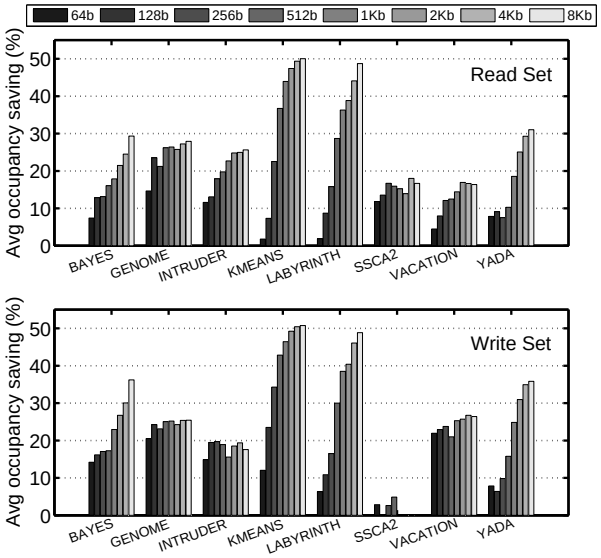

Fig. 5. Average Bloom filter occupancy saving of locality-sensitive signature with respect to generic signature, both for read set (top) and write set (bottom), varying the filter size from 64b to 8Kb.

D. Locality in benchmarks

Prior to including locality-sensitive signatures in a cycle accurate HTM simulator, a straightforward functional TM system was developed to estimate the locality properties of the benchmarks used in evaluating the present proposal (see Section IV-B). Intel's PIN instrumentation tool [43] was used to rapidly implement the system. PIN intercepts the execution of the first instruction of a program, and instruments it dynamically. To implement the TM system, routines were developed that analyze 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 the transaction. When a "commit_xact()" instruction is found, the transaction commits.

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

Using PIN TM simulator, the filter occupancy (that is, the number of bits set to 1) of committed transactions was recorded for both generic and LS signatures. The av-

erage occupancy saving percentage of LS signatures, with respect to generic ones, was calculated as $saving = \frac{occupancy\_generic - occupancy\_locality}{occupancy\_generic}$ and shown in Fig. 5. The results were obtained by varying the filter size from 64Kbit to 8Kbit. On large filters, occupancy conflicts are rather unlikely; therefore, rightmost bars show the maximum possible occupancy savings that LS-Sig can achieve for the tested benchmarks. It is to be noted that codes with the highest locality in Table II get the highest savings. On the other hand, SSCA2 benchmark hardly saves bits in the signatures because its transactions are very small. In fact, its WS is of 1 or 2 addresses on average (see Table III) and the saving is 0% as shown in Fig. 5. Finally, occupancy saving helps in diminishing the probability of stalling or aborting transactions due to false positives. Hence, these significant occupancy savings of LS-Sig are expected to improve the execution time.

IV. Experimental Evaluation

This section deals with the simulation environment and methodology (Section IV-A), the STAMP benchmark suite used for evaluating the present proposal (Section IV-B) and the experimental results obtained from the simulator (Sections IV-C to IV-F). Section IV-C explores different LS-Sig schemes, Section IV-D discusses about false sharing and LS-Sig, Section IV-E describes the effect of applying PBX hashing on LS-Sig, and finally, Section IV-F discusses about saving hardware.

A. Simulation environment and methodology

The simulation environment comprises a full system execution-driven simulator called Simics [44] along with the HTM module GEMS [29] provided by the Wisconsin Multifacet Project as open-source.

Simics simulates the SPARC architecture and runs an unmodified copy of a Solaris operating system. Solaris 10 was installed on the simulated machine and all workloads run on its top. GEMS's Ruby module, which includes a detailed timing model for the memory system, implements the LogTM-SE HTM [11]. Ruby was modified to include the locality-sensitive signature designs. For implementing the hash functions, same H3 matrices of Ruby were used after effecting the modifications described in Section III-A.

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

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

Finally, Ruby adds pseudorandom delays to the latency of memory accesses to deal with variability in simulation experiments. Therefore, each experiment was carried out multiple times to obtain confident error bars [45].

B. Benchmarks

All the benchmarks used in this paper belong to the Stanford's STAMP suite [28]. This suite is designed for Transactional Memory research and includes a wide range of applications, particularly those with long-running transactions and large read and write sets. Such benchmarks are of special interest for signature evaluation, because they put the most pressure on signatures. STAMP benchmarks were adapted to GEMS by applying Luke Yen's patches from the University of Wisconsin, Madison.

Specifically, the patches introduce the following changes in 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) the memory pools are traversed before starting computation to avoid page faults inside transactions; (iv) shared data structures are padded to avoid false sharing at cache block 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 are split to improve scalability for small signatures; (vii) in Labyrinth benchmark, the code that privatizes the grid is enclosed in open transaction to avoid inserting those reads in the signature. For effecting 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 workloads. Column "#xact" shows the number of committed transactions, and the column "Time in xact" the percentage of transactional cycles. The metric for locality in benchmarks, column "Xact locality", was drawn from Table II. The last four columns stand for average and maximum values of RS and WS size distributions in cache blocks.

C. Exploring radius and delta

Some works [30][46] consider that parallel signatures should be used, instead of the regular ones, because they can perform equally well or even better, besides being more area-efficient. Another suggestion they offered is to use four or more high-quality hash functions, preferably from the H3 family. The present experiments also prove (not shown) that, in most cases, the performance given by four hash functions is better than that given by one or two functions. Besides, using eight hash functions gives no significant improvement; instead it requires additional hardware. Consequently, all experiments were carried out using parallel Bloom filters with 4 H3 hash functions.

Following Def. 1, six (r, δ)-LS signatures were explored by combining two different delta functions, δ_0 and δ_1, with three

[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 TRANSACTIONAL CHARACTERISTICS. (DATA SET SIZES ARE REPORTED IN CACHE BLOCKS.)

Bench	Input	#xact	Time in xact	Xact locality	*avg* |*RS*|	*avg* |*WS*|	*max* |*RS*|	*max* |*WS*|
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

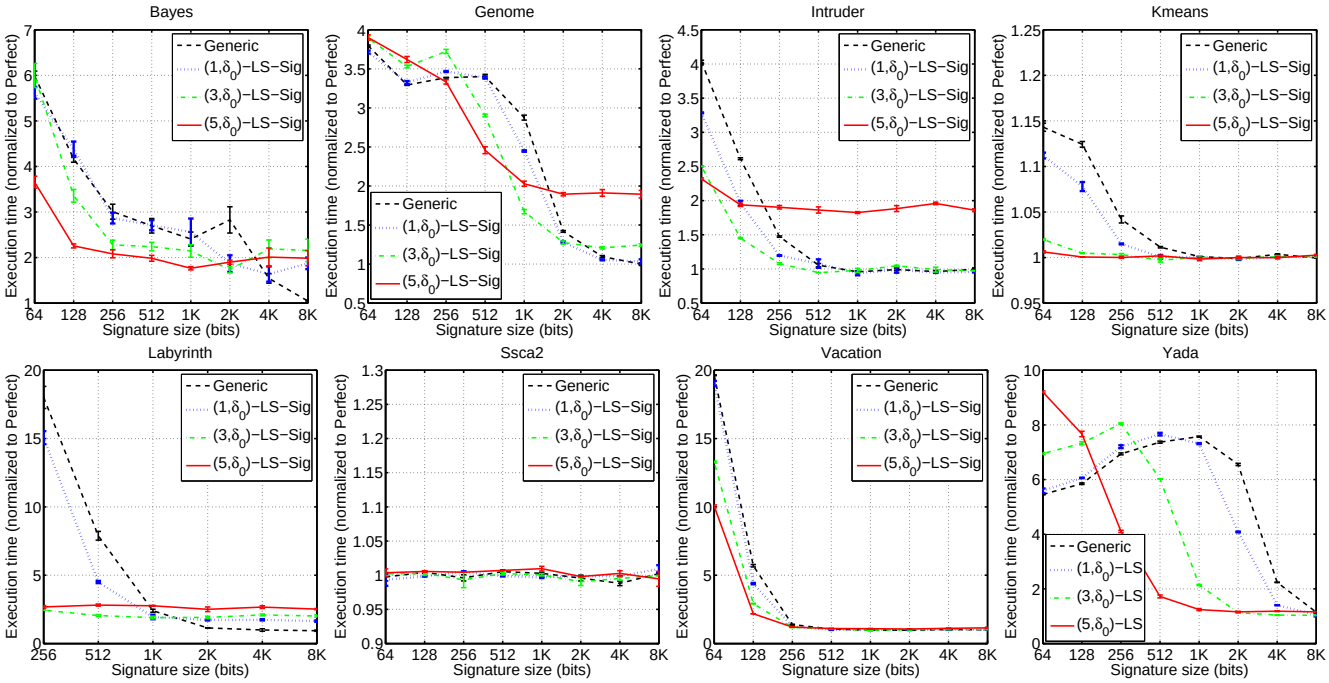

Fig. 6. Execution time normalized to perfect signature comparing parallel generic signatures and (r, δ)-LS-Sig with $r \in \{1, 3, 5\}$ and δ_0.

different radii, 1, 3 and 5. Also, another delta function, δ_P, is proposed, which behaves better independently of benchmarks. Its radii are 3 and 5. So, the following are the explored LS schemes:

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

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

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

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

3) (r, δ_P)-LS: Addresses within intervals of radius r were mapped depending on the distance between them. Thus, delta function is defined piecewise as follows:

$$\delta_P(d(x, y)) = \begin{cases} 1 & \text{if } d(x, y) = 1 \\ 2 & \text{if } 2 \leq d(x, y) \leq 2^{\lceil \frac{r}{2} \rceil} - 1 \\ 3 & \text{if } 2^{\lceil \frac{r}{2} \rceil} \leq d(x, y) \leq 2^r - 1 \end{cases}$$

For example, for $r = 5$, δ_P is as follows:

$$\delta_P(d(x, y)) = \begin{cases} 1 & \text{if } d(x, y) = 1 \\ 2 & \text{if } 2 \leq d(x, y) \leq 7 \\ 3 & \text{if } 8 \leq d(x, y) \leq 31 \end{cases}$$

Fig. 6 shows the execution time normalized to perfect filters for all the benchmarks using locality-sensitive signatures with δ_0 and three different radii. Time for parallel generic filters is also shown for comparison.

(r, δ_0)-LS-Sig operates as a generic filter which maps the addresses at granularity higher than that of cache blocks. Thus, δ_0 could suffer from additional false sharing (see Section IV-D) when the radius is high. For radius equal to 1, the memory block is 1 bit larger from the viewpoint of signatures. It introduces little additional false sharing and consequently almost all benchmarks behave the same than parallel generic signatures do when the filters are large, except for Labyrinth and Bayes, which performs worse, because of false sharing, with about $1.7\times$ slowdown for 2Kb, 4Kb and 8Kb in Labyrinth and $1.7\times$ slowdown for 8Kb in Bayes. With shorter filters, $(1, \delta_0)$-LS-Sig performs equally well or slightly better due to occupancy saving. For radius equal to

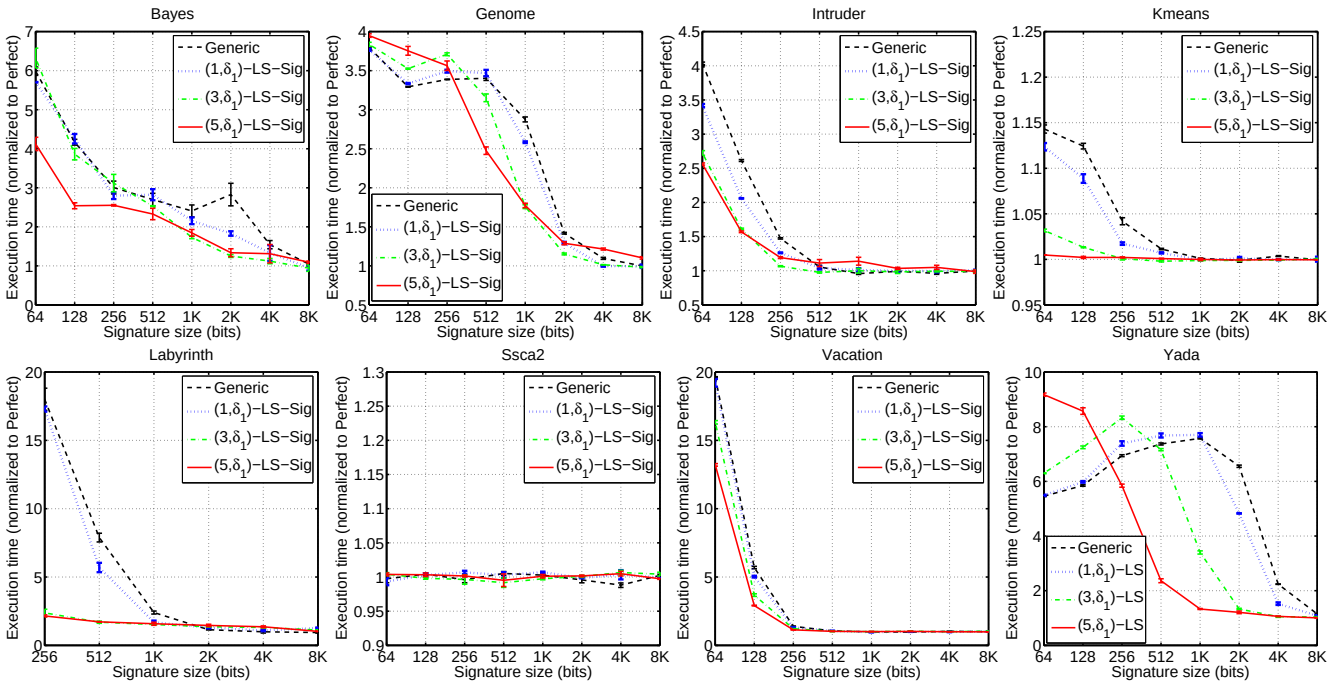

Fig. 7. Execution time normalized to perfect signature comparing parallel generic signatures and (r, δ)-LS-Sig with $r \in \{1, 3, 5\}$ and δ_1.

3, the signature block is larger by 3 bits; so, there is more additional false sharing, as shown by larger filters. It can be seen from Fig. 6 that Labyrinth slows down by $2\times$ for 4Kb and 8Kb signatures, Bayes by $2\times$ for 8Kb signature, and Genome by $1.25\times$ for 8Kb signature. Conversely, occupancy savings lead to better results when the signature size decreases: Bayes shows a speedup of $1.25\times$ from 128b to 2Kb signatures, Genome $1.7\times$ for 1Kb, and Intruder and Vacation $1.5\times$ on average for small signatures, while Labyrinth and Yada speed up by $7\times$ and $5\times$ respectively. When the radius is equal to 5, 2^5 addresses were mapped to the same four bits in the signature. Consequently, false sharing results in $2\times$ slowdown of large signature performance for Bayes, Genome, Labyrinth and Intruder. However, the sparse filling of the filter leads to significant results for almost every benchmark. In most cases, Labyrinth, Yada, Bayes, Intruder and Vacation are twice as fast as parallel generic signatures, as shown in Fig. 6, whereas Yada and Labyrinth are six-times as fast in some cases.

Fig. 7 shows the execution time normalized to perfect filters for all the benchmarks using (r, δ_1)-LS-Sig with $r \in \{1, 3, 5\}$. δ_1 combines three hash functions operating at granularity coarser than that of cache and one hash function working at cache block granularity. This way, the hash function working at cache granularity can distinguish between addresses that map to the same bits because of coarser granularity of the other three hash functions, as long as it does not incur a false positive. As Fig. 7 shows, benchmark performance for large signature sizes is now either slightly worse than parallel signatures or the same. With $r = 5$, Genome slows down its execution by $1.1\times$ for 4Kb and 8Kb signatures, Intruder by $1.2\times$ for 1Kb and Labyrinth $1.36\times$ for 4Kb. For small signatures, execution is slower than that of δ_0, because of the addition of false positives from the hash function that operates at cache granularity in addition to the false sharing from the three hash functions operating at coarser granularity. Even so, δ_1 outperforms parallel signatures in most cases.

Additionally, one more LS scheme was defined that performs more evenly, whatever be the benchmark: (r, δ_P)-LS. Such a scheme merges several granularities to trade off between additional false sharing and false positives. Fig. 8 shows the execution time normalized to perfect signatures for parallel generic signatures, (r, δ_P)-LS-Sig with $r \in \{3, 5\}$ and $(5, \delta_1)$-LS-Sig. It is to be noted that (r, δ_P)-LS-Sig performs the same way or better than parallel filters do on large signature sizes so that they can get rid of the additional false sharing effect that comes up in δ_1 and δ_0 for Bayes, Genome, Intruder or Labyrinth. As far as small signatures are concerned, $(3, \delta_P)$-LS-Sig gets closer to parallel filters because it is built up with smaller radius than that of $(5, \delta_P)$-LS-Sig, which, conversely, gets closer to $(5, \delta_1)$-LS-Sig.

From the results of (r, δ_P)-LS-Sig, shown in Fig. 8, benchmarks can be classified into three groups, considering their behavior:

1) *SSCA2:* This benchmark exhibits the smallest transactions of the whole suite. RS and WS maximum sizes are only 3 and 2 cache blocks respectively. Moreover, the benchmark spends most of the time outside transactions (see Table III). Hence, SSCA2 is not signature-dependent.
2) *Kmeans, Vacation, Intruder and Labyrinth:* These benchmarks show similar behavior when signature size decreases. In some cases, $(3, \delta_P)$-LS-Sig and $(5, \delta_P)$-LS-Sig reduce the execution time considerably. They always either outperform or match the performance of generic signatures.

   Kmeans is low contended and spends only 6% of time

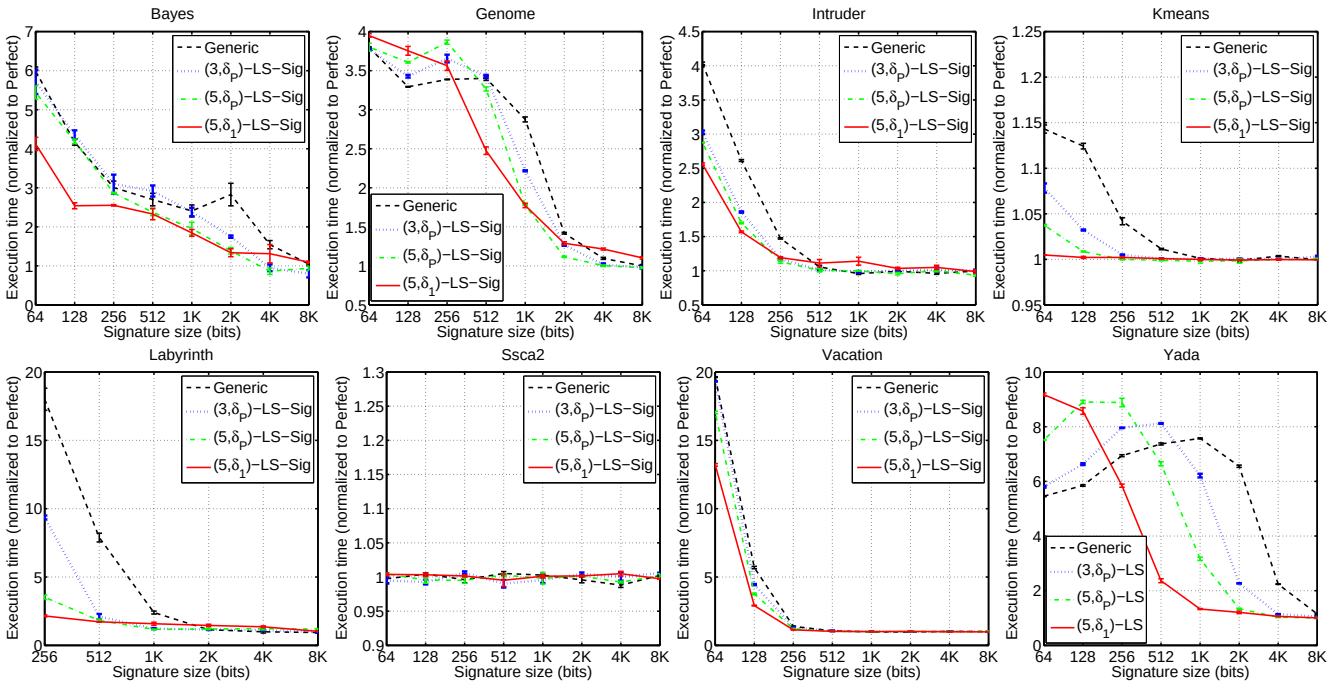

Fig. 8. Execution time normalized to perfect signature comparing parallel generic signatures and (r, δ)-LS-Sig with $r \in \{3, 5\}$ for δ_P and $r = 5$ for δ_1.

in transactions; so, this is also not signature-dependent (Fig. 8 shows a speedup of 1.14x for the best case). Even so, (r, δ_P)-LS-Sig reduces execution time of generic ones when 128b filters are used, because transactions are of medium size and exhibit high locality (see Table II). In Vacation, (r, δ_P)-LS-Sig matches the execution time of generic signatures, because of mid-locality and medium-to-small scale transactions. They are better for 64 and 128b signatures, because the maximum data set size (max RS size is 90) is close to the signature size, and generic signatures have higher occupancy. Vacation is high contended and does not scale well for small signatures. Intruder behaves the same way as does Vacation, but scales better.

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

3) *Bayes, Genome, Yada:* In these benchmarks, LS-Sig yields better results for large signature sizes than generic signatures do, but slightly worse results for small ones. This behavior is related to the manner in which LogTM-SE resolves conflicts. LogTM-SE stalls transactions that request for a conflicting address, retries its coherence operation, and aborts on a possible deadlock cycle. Hence, with (r, δ_P)-LS-Sig, transactions can run for a longer time before encountering a conflict, even on small signatures, but on abort, they must undo the log that is longer than what it was, had the conflict been detected earlier. Fig. 9 shows the average RS and WS percentages of false positives. The percentage of false positives was obtained by dividing the total number of false positives by the total number of both false and true positives.

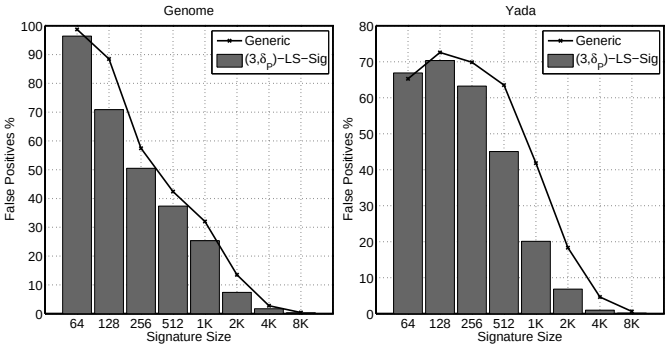

Fig. 9. Average of RS and WS percentage of false positives for generic and $(3,\delta_P)$-LS signatures.

Fig. 8 shows that the number of false positives is higher for signatures from 64b to 256b, but Fig. 9 shows that the number of true positives also is higher as the percentage becomes low. This confirms that transactions run longer by virtue of (r, δ_P)-LS-Sig. It is to be noted that decreasing false positive rate in signatures does not necessarily lead to direct improvement in performance, as other factors, like abort patterns, also matter.

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

Finally, the linear correlation between execution time and abort time was calculated for the experimental setups of Figs. 6, 7 and 8. The total time of aborts depends on the number of aborts, as also of the time required to solve them, which is mainly a function of the length of the log to be

undone. The observed correlation coefficient varies from 0.88 to 0.99, which reflects the strong correlation between the execution time and the abort time.

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

D. Interthread vs. intrathread locality: the effect of additional false sharing

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

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

- δ_0*, every hash function at the same granularity larger than cache blocks*: An additional false sharing false positive is detected if the address involved was not really inserted in the filter, but another address was inserted within the same signature super block. The filter thus matches the address because of additional false sharing.
- δ_1*, one hash function at cache block granularity and the others at the same larger granularity*: A false positive due to additional false sharing is detected if there is a false positive due to aliasing/occupancy in the filter which operates at cache block granularity, and the address was not really inserted, but another one in its super block was.
- δ_P*, one hash function at cache granularity and the others at larger and different granularities*: A false positive due to additional false sharing is detected if there is a false positive due to aliasing/occupancy in the filter which operates at cache block granularity, and all the other filters operating at different and coarser granularity result in a false sharing false positive.

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

Fig. 10 shows the number of total false positives (FP), i.e. the number of false positives in the read set of all transactions plus the number of false positives in the write set of all transactions for parallel generic signatures and, the number of false positives for $(5, \delta_0)$-LS-Sig, $(5, \delta_1)$-LS-Sig and $(5, \delta_P)$-LS-Sig. False positives concerning LS-Sig were broken down into false positives due to signature aliasing/occupancy (FP) and those due to signature false sharing (FS). The figure shows five benchmarks that are sensitive to signature false sharing: Bayes, Genome, Intruder, Labyrinth and Yada.

It needs to be noted that δ_0, the top row, is the locality-sensitive hash function that shows more false positives due to additional false sharing. In this case, when the filter size is large, the number of false positives due to occupancy is close to zero and almost every false positive is due to false sharing (light gray in Fig. 10). This situation leads to the definition of two types of locality in a parallel benchmark and therefore, to the classification of benchmarks depending on the type of locality they exhibit:

- *Interthread locality*: A parallel code shows interthread locality, if their threads reference shared memory locations near in time and there is spatial proximity between such locations.
  Fig. 6 shows that Bayes and Labyrinth slow down their execution for $(r \in \{1, 3, 5\}, \delta_0)$-LS 8Kb signatures, implying thereby that such benchmarks exhibit high interthread locality. Genome and Intruder get worse results with radius 5, but with radius 1 or 3 they do not slow down their execution significantly. Thus, shared data is located farther away from each other than that of Bayes and Labyrinth; so, it can be said that Genome and Intruder exhibit medium interthread locality. The other benchmarks, which do not show significant slowdown for $(5, \delta_0)$-LS-Sig, can be regarded as of low interthread locality. Overall, interthread locality is likely to be a feature of codes whose shared data comprise arrays of primitive data types and light-weight structures, like the Labyrinth's three dimensional integer array which represents the maze where the benchmark finds the shortest-distance paths between pairs of points, or the net of structs which holds the marks of the Bayesian network in Bayes. However, the shared data access patterns will ultimately determine the degree of interthread locality in a parallel application.
- *Intrathread locality*: A parallel code shows intrathread locality if their threads reference their private memory locations near in time and there is spatial proximity between locations.

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

The middle row of Fig. 10 shows the total number of additional false sharing false positives for $(5, \delta_1)$-LS-Sig, which has a filter that operates at cache granularity. It avoids false positives due to false sharing in other filters as long as a false positive due to occupancy is not detected in that filter. If it is detected, the other filters may get a false positive due to false sharing. The probability of getting false positives increases with the occupancy of the filter; so, the probability of getting false positives due to false sharing will be higher when the first filter gets full. However, thanks to locality-sensitive signatures, three more filters, operating at higher granularity, have not been spoiled by occupancy. Furthermore, by the time the first filter has been spoiled by occupancy, there will be lots of addresses in the signature that may probably form blocks of contiguous, locality-wise addresses, so that the other filters that have been storing super block addresses do not get lots of false positives due to false sharing, because almost every

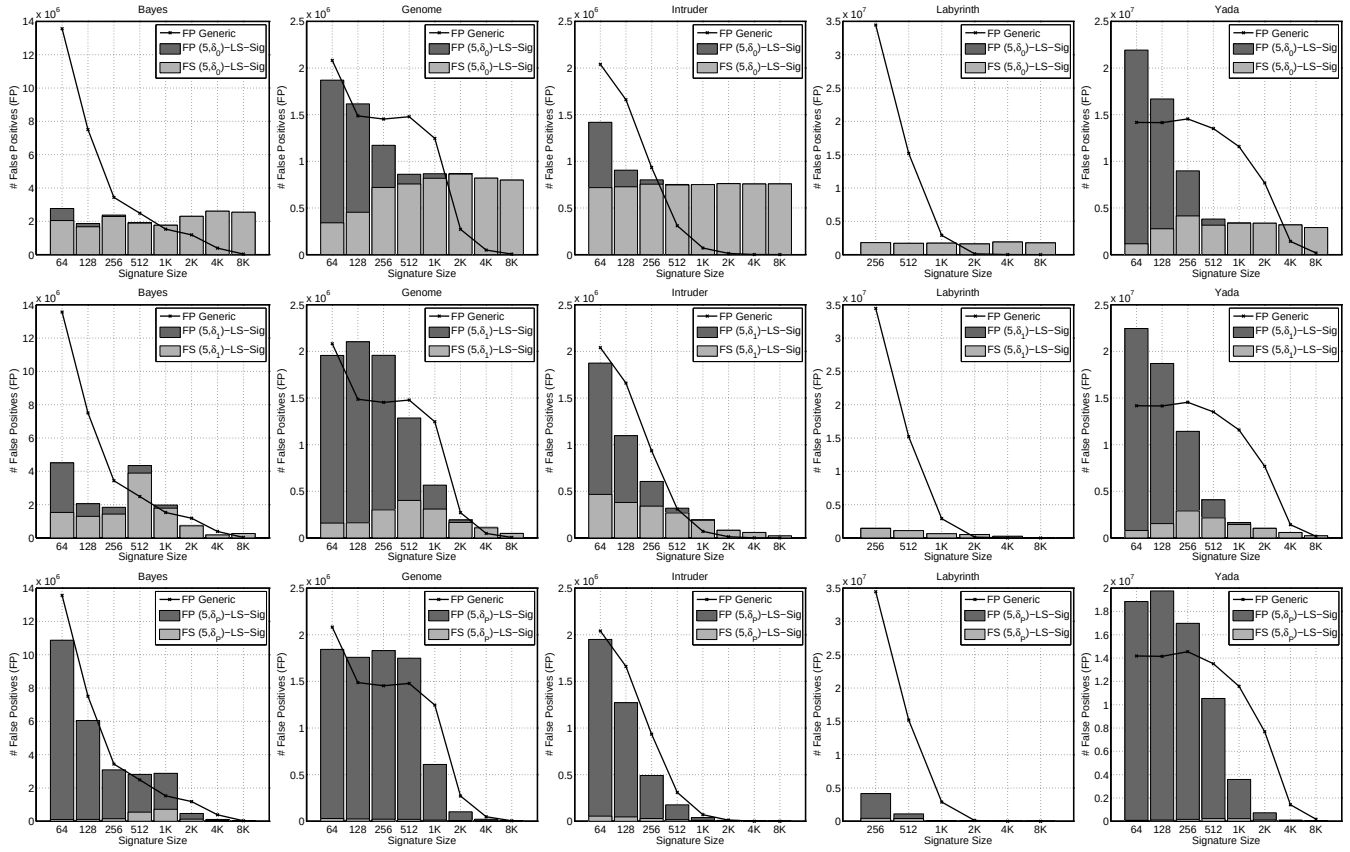

Fig. 10. Number of false positives of parallel generic signatures (FP Generic) compared to the number of false positives of $(5, \delta_0)$-LS-Sig (top row), $(5, \delta_1)$-LS-Sig (middle row) and $(5, \delta_P)$-LS-Sig (bottom row). False positives for LS-Sig are broken down into false positives due to aliasing (FP) and false positives due to additional false sharing (FS).

address in a super block will probably have been inserted. This is better achieved by functions of class δ_P, because they map addresses at different granularities, both fine and coarse. Such a scheme minimizes the number of false conflicts due to false sharing as shown by the bottom row graphs in Fig. 10.

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

E. LS-Sig PBX hashing

In this section, Page-Block-XOR (PBX) hashing [33] was used to reduce hardware costs of Locality-sensitive Signatures. PBX hashing was devised by Yen et al. to keep the randomness of H3 functions while saving in area, power and latency of the signature implementation. H3 hashing requires a tree of XOR gates per bit of the hashing function output index, conversely, PBX requires only one XOR gate per bit resulting in a single XOR gate row per hashing function.

The insight behind PBX is that the input bits have enough randomness to minimize the XOR trees involved in the index computation. Such randomness was calculated using the entropy of the addresses. Assuming 32-bit virtual and physical addresses, cache-blocks of 64 bytes and page size of 4kB, higher entropy (i.e. higher randomness) was found from the address bit, 26th to the 6th of the addresses. Yen et al. also found that bit-field overlap leads to higher false positive due to correlation between bit-fields. Therefore, the physical page number bits, from bit 26th to bit 17th, were combined with the cache-index bits[2], from bit 16th to bit 6th. The Appendix shows the PBX binary matrices used in this paper. Such matrices define a surjection, that is, for every value into the codomain of indexes there exists at least one value in the domain of addresses. To assure that, the rank in $GF(2)$ (the Galois field of two elements) of the matrices was calculated and proved to have full rank. To get the LS-Sig PBX matrices, they were shifted downwards inserting blank rows on the top, thus maintaining the bit-field disjunction and ensuring maximum use of bits. However, as signature gets smaller, the leftmost columns were subtracted from the matrices, keeping the bits in those columns from being used to compute the final index. This way, LS-Sig PBX could yield slightly different results for some benchmarks (e.g. Yada, Labyrinth) when signatures are small as shown in Fig. 11.

While working with Bloom filters implemented as regular filters, it needs to be ensured that the union of matrices by

[2]The size of the bit-fields are not tied to the configuration of the CMP [33]

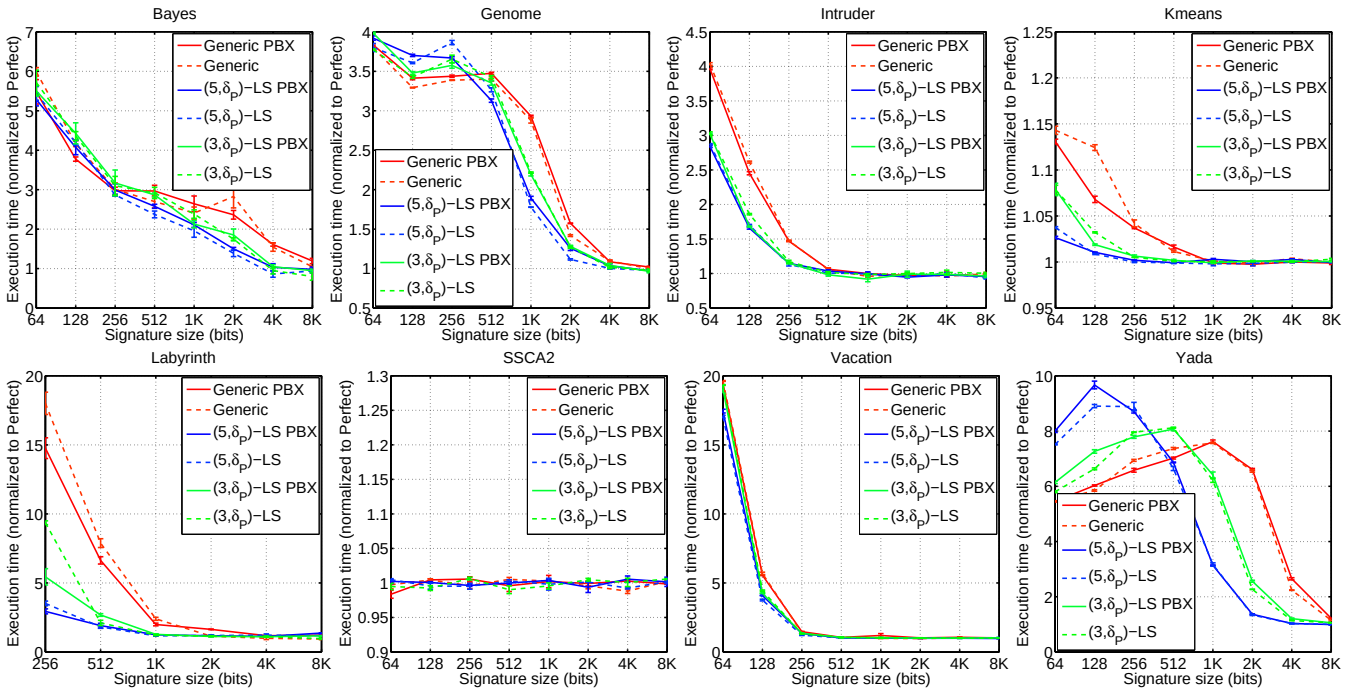

Fig. 11. Execution time normalized to perfect signature (no false positives) comparing Generic and $(r \in \{3,5\}, \delta_P)$-LS-Sig to their PBX versions.

pairs is of full rank. This way, all hash functions will yield different indexes for a given address. In this paper, parallel implementation was used so that there will be no need to check the rank of the union of the matrices. Hash functions may yield the same index for a given address because different subarrays are asserted. Even so, the matrices shown in the Appendix have full rank by pairs.

F. Saving hardware

LS-Sig can be considered to enable smaller signature sizes, as opposed to improving only false-conflicts. Fig. 8 shows that Yada and Labyrinth yield the same results if a parallel LS-Sig is used with half the size of generic ones. Intruder and Genome behave similarly, but when the signature is halved, they behave slightly worse. Vacation and Bayes could be in the same group as Yada and Labyrinth, but only when the signature size is not halved from 256 bit downward.

As regards the hash functions, according to the XOR count proposed by Yen et al. [33], $(5, \delta_P)$-LS-Sig reduces the number of XOR gates by about 6.5% with respect to the generic version. On the other hand, PBX can achieve a reduction of up to 80%. However, the design of PBX signatures is subject to prior analysis of the entropy of the workloads. It is to be noted that the area required by hash functions may represent about one-fifth the size of the SRAM for $k = 4$ [30].

V. Conclusions

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

LS-Sig was implemented in the Wisconsin GEMS simulator and was evaluated using the STAMP benchmark suite. The results obtained show that the proposed signature improves benchmark performance in most cases, especially in large set long-running transactions, where the probability of false positives may be high. A remarkable feature of LS-Sig is that it does not require additional hardware over what is needed for generic signatures. Therefore, parallel locality-sensitive signatures are a good alternative to generic ones in that they yield similar or even better performance at the same cost.

Acknowledgment

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

References

[1] 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.

[2] T. Harris, J. Larus, and R. Rajwar, *Transactional Memory, 2nd edition*. Morgan & Claypool Pub., 2010.

[3] 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.

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

[5] L. Hammond *et al.*, "Transactional memory coherence and consistency," in *31th Ann. Int'l. Symp. on Computer Architecture (ISCA'04)*, 2004, pp. 102–113.

[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] 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 an Operating Systems (ASPLOS'09)*, 2009, pp. 157–168.
[8] C. Blundell, J. Devietti, E. Christopher Lewis, and M. Martin, "Making the fast case common and the uncommon case simple in unbounded transactional memory," in *34th Ann. Int'l. Symp. on Computer Architecture (ISCA'07)*, 2007, pp. 24–34.
[9] W. Chuang *et al.*, "Unbounded page-based transactional memory," in *12th Int'l. Conf. on Architectural Support for Programming Language an Operating Systems (ASPLOS'06)*, 2006, pp. 347–358.
[10] 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.
[11] L. Yen *et al.*, "LogTM-SE: Decoupling hardware transactional memory from caches," in *13th Int'l. Symp. on High-Performance Computer Architecture (HPCA'07)*, 2007, pp. 261–272.
[12] J. Bobba, N. Goyal, M. Hill, M. Swift, and D. Wood, "TokenTM: Efficient execution of large transactions with hardware transactional memory," in *35th Ann. Int'l. Symp. on Computer Architecture (ISCA'08)*, 2008, pp. 127–138.
[13] S. Tomic *et al.*, "EazyHTM: Eager-lazy hardware transactional memory," in *42nd IEEE/ACM Ann. Int'l. Symp. on Microarchitecture (MICRO'09)*, 2009, pp. 145–155.
[14] S. Wang *et al.*, "DTM: Decoupled hardware transactional memory to support unbounded transaction and operating system," in *38th Int'l. Conf. on Parallel Processing (ICPP'09)*, 2009, pp. 228–236.
[15] S. Jafri, M. Thottethodi, and T. Vijaykumar, "LiteTM: Reducing transactional state overhead," in *16th Int'l. Symp. on High-Performance Computer Architecture (HPCA'10)*, 2010, pp. 81–92.
[16] P. Damron *et al.*, "Hybrid transactional memory," in *12th Int'l. Conf. on Architectural Support for Programming Language an Operating Systems (ASPLOS'06)*, 2006, pp. 336–346.
[17] S. Kumar, M. Chu, C. Hughes, P. Kundu, and A. Nguyen, "Hybrid transactional memory," in *11th ACM Symp. on Principles and Practice of Parallel Programming (PPoPP'06)*, 2006, pp. 209–220.
[18] Y. Lev, M. Moir, and D. Nussbaum, "PhTM: Phased transactional memory," in *2nd Workshop on Transactional Computing (TRANSACT'07)*, 2007.
[19] B. Saha, A.-R. Adl-Tabatabai, and Q. Jacobson, "Architectural support for software transactional memory," in *39st IEEE/ACM Ann. Int'l. Symp. on Microarchitecture (MICRO'06)*, 2006, pp. 185–196.
[20] A. Shriraman *et al.*, "An integrated hardware-software approach to flexible transactional memory," in *34th Ann. Int'l. Symp. on Computer Architecture (ISCA'07)*, 2007, pp. 104–115.
[21] S. Shriraman, S. Dwarkadas, and M. Scott, "Flexible decoupled transactional memory support," in *35th Ann. Int'l. Symp. on Computer Architecture (ISCA'08)*, 2008, pp. 139–150.
[22] 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.
[23] C. Minh *et al.*, "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.
[24] M. Lupon, G. Magklis, and A. Gonzalez, "FASTM: A log-based hardware transactional memory with fast abort recovery," in *18th Int'l. Conf. on Parallel Architectures and Compilation Techniques (PACT'09)*, 2009, pp. 293–302.
[25] ——, "A dynamically adaptable hardware transactional memory," in *43rd IEEE/ACM Ann. Int'l. Symp. on Microarchitecture (MICRO'10)*, 2010.
[26] 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 Languages Design and Implementation (PLDI'09)*, 2009, pp. 166–176.
[27] B. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.
[28] 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.
[29] M. Martin *et al.*, "Multifacet's general execution-driven multiprocessor simulator GEMS toolset," *ACM SIGARCH Comput. Archit. News*, vol. 33, no. 4, pp. 92–99, 2005.
[30] 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.
[31] L. Carter and M. Wegman, "Universal classes of hash functions," *J. Computer and System Sciences*, vol. 18, no. 2, pp. 143–154, 1979.
[32] 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.
[33] 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.
[34] W. Choi and J. Draper, "Locality-aware adaptive grain signatures for transactional memories," in *IEEE Int'l. Symp. on Parallel and Distributed Procesing (IPDPS'10)*, 2010, pp. 1–10.
[35] M. Labrecque, J. M., 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.
[36] H. Vandierendonck and K. De Bosschere, "XOR-based hash functions," *IEEE Trans. on Computers*, vol. 54, no. 7, pp. 800–812, 2005.
[37] J. Chung *et al.*, "The common case transactional behavior of multithreaded programs," in *12th Int'l Symp. on High Performance Computer Architecture (HPCA'06)*, 2006, pp. 266–277.
[38] M. Charikar, "Similarity estimation techniques from rounding algorithms," in *34th Ann. ACM Symp. on Theory of Computing (STOC'02)*, 2002, pp. 380–388.
[39] 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.
[40] A. Kirsch and M. Mitzenmacher, "Distance-sensitive bloom filters," in *8th Workshop on Algorithm Engineering and Experiments (ALENEX'06)*, 2006, pp. 41–50.
[41] I. Kotera, R. Egawa, H. Takizawa, and H. Kobayashi, "Modeling of cache access behavior based on Zipf's law," in *Proceedings of the 9th workshop on memory performance: dealing with applications, systems and architecture*, 2008, pp. 9–15.
[42] P. J. Denning, "The locality principle," Naval Postgraduate School, Monterey, California, Tech. Rep., 2008.
[43] C.-K. Luk *et al.*, "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.
[44] P. Magnusson *et al.*, "Simics: A full system simulation platform," *IEEE Computer*, vol. 35, no. 2, pp. 50–58, 2002.
[45] 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.
[46] R. Quislant, E. Gutierrez, O. Plata, and E. L. Zapata, "Improving signatures by locality exploitation for transactional memory," in *Parallel Architectures and Compilation Techniques, International Conference on*, 2009, pp. 303–312.

Ricardo Quislant received the MSc degree in Computer Engineering from the University of Granada, Spain, in 2006. He is currently working toward the PhD degree in the Department of Computer Architecture, University of Málaga. His main research interests are computer memory system and high performance computing, with special regard to transactional memory.

Eladio Gutiérrez received the MSc and PhD degrees in Telecommunication Engineering from the University of Málaga, Spain, in 1995 and 2001, respectively. He is currently an associate professor in the Department of Computer Architecture at the University of Málaga. His research interests include parallel architectures, graphics processing units and automatic parallelization.

Oscar Plata received the MSc and PhD degrees in Physics from the University of Santiago de Compostela, Spain, in 1985 and 1989, respectively. Currently, he is a full professor in the Department of Computer Architecture at the University of Málaga, Spain. His research interests include high-performance computing and compiler techniques for parallel architectures.

Emilio L. Zapata received the MSc degree in Physics from the University of Granada in 1978 and the PhD degree in Physics from the University of Santiago de Compostela, Spain, in 1983. Since 1991, he has been a full professor in the Department of Computer Architecture at the University of Málaga. His main research interests are numerical and audiovisual applications, high performance architectures and compilation techniques for parallel computers.