Scalability Analysis of Signatures in Transactional Memory Systems

Ricardo Quislant, Eladio Gutierrez, Oscar Plata
Dept. of Computer Architecture
University of Malaga, Spain
Email: {quislant, eladio, oplata}@uma.es

***Abstract*—Signatures have been proposed in transactional memory systems to represent read and write sets and to decouple transaction conflict detection from private caches or to accelerate it. Generally, signatures are implemented as Bloom filters that allow unbounded read/write sets to be summarized in bounded space at the cost of false conflict detection. It is known that this behavior has great impact in parallel performance.**

In this work, a scalability study of state-of-the-art signature designs is presented, for different orthogonal transactional characteristics, including contention, length, concurrency and spatial locality. This study was accomplished using the Stanford EigenBench benchmark. This benchmark was modified to support spatial locality analysis using a Zipf address distribution. Experimental evaluation on a hardware transactional memory simulator shows the impact of those parameters in the behavior of state-of-the-art signatures.

Keywords*—*Hardware transactional memory, Bloom filter, signatures, conflict detection, locality, multiset, asymmetric

I. Introduction

Transactional Memory (TM) [1] has emerged as an alternative to the conventional multithread programming to ease the writing of concurrent programs. TM introduces the concept of transaction that allows to separate atomicity and isolation semantics from implementation. In order to implement the transaction abstraction effectively, the TM system has to keep track of the data read and written by the transactions running in the system, for the system to accordingly detect and act on a possible conflict between them.

Signatures have been proposed recently to store the addresses of such memory reads and writes to decouple transaction conflict detection from private caches or to accelerate it. Basically, signatures are implemented as Bloom filters [2], structures that use fixed space to summarize an unbounded amount of read/write memory addresses. The price of this implementation is that there is a possibility of detecting false conflicts, that is, non-existing conflicts that can have a great impact in parallel performance.

In this work, a scalability study of state-of-the-art signature designs is presented, for different orthogonal transactional characteristics, including contention, length, concurrency and spatial locality. This study was accomplished using the Stanford EigenBench benchmark, a microbenchmark that can emulate a set of orthogonal application characteristics. We modified the benchmark to support spatial locality analysis using a Zipf address distribution.

The experimental evaluation was carried out in a hardware transactional memory (HTM) simulator where we implemented the state-of-the-art enhanced signature schemes that were tested with a fixed signature size throughout the different application characteristics. Results show that enhanced signatures improve the overall performance of the system and can be thought of a way to reduce space requirements as well. However, enhanced and conventional signatures converge when transactions are very large or contention in the TM system is too high. Conversely, enhanced signatures scale better than conventional ones when we increase concurrency in the system. Other factors like conflict detection granularity and spatial locality improve the performance of signatures.

The remainder of the paper is organized as follows. Next section discusses a background on signatures describing the main signature schemes analyzed in this work. Section III describes the EigenBench benchmark and the modifications we have included. Section IV shows the experimental evaluation and all the results obtained with the simulator. Finally, we draw the conclusions in Section V.

II. Background

Ceze et al. [3] proposed *signatures* as a compact way of representing the read set (RS) and the write set (WS) of transactions by means of Bloom filters [2], a time and space-efficient hash structure. Since then, signatures have been broadly adopted by several software and hardware TM systems to detach conflict detection from caches or accelerate conflict detection [4], [5].

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

Bloom filters, also known as true or regular Bloom filters, are implemented as a k-ported SRAM, with k being the number of hash functions and the SRAM implementing a bit vector. Sanchez et al. [6] proposed the parallel Bloom filter as an alternative hardware-efficient implementation to regular Bloom filters. The parallel filter consists of k 1-ported SRAMs, and it yields the similar false positive rate.

Since the work of Sanchez et al., several works have been proposed to deal with false positives and to enhance signature performance. Locality-sensitive signatures (LS-Sig)

are proposed in [7] to exploit memory reference locality and reduce the probability of false conflicts. The proposal defines new maps for hash functions to reduce the number of bits inserted in the filter (occupancy) for those addresses with spatial locality. That is, nearby memory locations share some bits of the Bloom filter. They define several locality-sensitive hash functions with special interest in those defined piecewise, so-called (r, δ_P)-LS, where the subfilters in the parallel Bloom filter harness locality at different granularity, with a maximum value of r. Its implementation does not require extra hardware.

Multiset and reconfigurable asymmetric signatures [8] address the fact that transactions frequently exhibit read and write sets of uneven cardinality. Wheras conventional parallel signatures devote same-sized filters to each set, multiset (MS) ones comprise a single Bloom filter to track both RS and WS. On the other hand, reconfigurable asymmetric (ASYM) signatures can be configured to have a subfilters for the RS and $2k-a$ subfilters for the WS. The multiset signature is also combined with hash sharing, where s subfilters share the same has function and $k-s$ still have separate RS and WS hashes.

Choi and Draper propose Unified signatures [9], that are the same as MS $s = k$ signatures. However, they propose augmenting the signature with an extra register to filter out read-read dependencies, since they share all hash functions, which unables to distinguish between read and written locations. The same helper register effect is achieve with multiset signatures by setting $s = k - 1$. Both Unified and MS signatures can be conbined with LS-Sig to glean more performance gain.

III. The EigenBench Benchmark

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

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

A. Modifications to EigenBench

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

EigenBench is released to work with TL2 [10], an STM system where transactional accesses must be explicitly annotated. The `TM_READ` and `TM_WRITE` instructions showed in Table I are used to do so. Hence, other non-annotated instructions are not tracked by the STM system. However, we use an implicit HTM system where all instructions enclosed by a transaction are implicitly taken as transactional. Then, calls to `random` function inside `rand_action` and `rand_index` functions are tracked by the transactional system provoking the serialization of transactions. To solve it, we used a Mersenne twister pseudorandom generator per thread which can be found in the library of the STAMP bencharmk suite [11]. Furthermore, to keep the TM system from tracking those implicit accesses we use escape actions [12].

EigenBench, as is, generates random memory traces that can be biased by the `lct` parameter to introduce temporal locality of reference with a given probability. We have modified the benchmark to include spatial locality of reference. We have defined the parameter `lcs` as the probability that an access is nearby located to a preceding access. For the spatial locality distribution we have used the notion of *random walk* introduced by Thiébaut et al. [13]. The sequential accesses of the program to memory can be modeled as a random walk through a one-dimensional integer array. This integer array is main memory, the walker is EigenBench, and the jumps correspond to the gaps between consecutive accesses. The length of each jump is a sample value of the random variable X with the following probability distribution:

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

where $u > 0$, and u_0 and θ are constants. The parameter θ describes the spatial locality of the random walk. As θ increases, the walk gets more locally distributed. We have chosen $\theta = u_0 = 1$, so that the random walk is governed by the simplest form of the Zipf distribution [14], where the first most common jump is of length $u = 1$, the second jump ($u = 2$) occurs $1/2$ as often as the first, the third most common jump ($u = 3$) occurs $1/3$ as often as the first and so on.

Table II shows the pseudocode of the modification to include the locality random walk. We have limited the jumps to a length of sixteen. Thus, jumps of length 1 have a probability of 0.3, while the probability of jumps of length 2 is 0.15, 0.1 for length 3, and so forth. To get such a Zipf distribution from a random distribution that equiprobably yields numbers between 0 and 1023, we have defined an array, in lines 2–3, with the boundaries of the intervals for each jump following the probabilities above. If the random number, in line 8, is lower than the first interval boundary, i.e. `rand` $\in [0, 303)$, then the jump is of length 1. If `rand` $\in [303, 454)$, the jump is of length 2, and so on. The length of the jump is calculated in lines 10–11. At the end of the `for` loop, the variable `jump` holds the length of the jump to be performed, so the following

TABLE I: Pseudocode of the EigenBench core function.

```
1  global long array_hot[N_HOT];
2  global long array_mild[N_MILD];
3  void test_core(tid, loops, lct, R_HOT, W_HOT, R_MILD, W_MILD, R_OUT, W_OUT) {
4    long val=0;
5    long total = W_HOT + W_MILD + R_HOT + R_MILD;
6    for (i=0; i<loops; i++) {
7      (r_hot, w_hot, r_mild, w_mild) = (R_HOT, W_HOT, R_HOT, W_MILD);
8      BEGIN_TM();
9      for (j=0; j<total ; j++) {
10       switch(rand_action(r_hot, w_hot, r_mild, w_mild)) {
11         case READ_HOT:
12           index = rand_index(tid, lct, array_hot);
13           val += TM_READ(array_hot[index]);
14         case WRITE_HOT:
15           index = rand_index(tid, lct, array_hot);
16           TM_WRITE(array_hot[index], val);
17         case READ_MILD:
18           index = rand_index(tid, lct, array_mild);
19           val += TM_READ(array_mild[index]);
20         case WRITE_MILD:
21           index = rand_index(tid, lct, array_mild);
22           TM_WRITE(array_mild[index], val);
23       }
24     }
25     END_TM();
26     val += local_ops(R_OUT, W_OUT, val, tid);
27   }
28 }
29 action rand_action(r_hot, w_hot, r_mild, w_mild) {
30 // With uniform random probability based on r_hot, w_hot, r_mild, w_mild
31 // randomly choose one among: READ_HOT, WRITE_HOT, READ_MILD, WRITE_MILD.
32 // Then, decrease corresponding variable (r_hot, r_mild,...) by one.
33 }
34 long rand_index(tid, lct, array) {
35 // With lct probability, choose a saved index from the history buffer, or
36 // randomly choose an index from range [0, N_HOT-1] or [tid*N_MILD,
37 // (tid+1)*N_MILD-1] and save it to the history buffer.
38 }
39 long local_ops(r_out, w_out, val, tid) {
40 // Perform r_out reads and w_out writes on a private array in random order.
41 }
```

TABLE II: Pseudocode of the function that generates the locality random walk.

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

lines get the last accessed location from the history buffer, and the jump is added to it, thus forming the new location to be accessed, which is inserted in the history buffer and then returned. Note that the jump can be randomly added to or subtracted from the last address accessed (lines 7 and 13).

B. Orthogonal TM Characteristics

EigenBench can be used to simulate a given execution pattern that exhibits a series of orthogonal TM characteristics. Hong et al. [15] define a set of *eigen-characteristics* that are orthogonal each other, but they can be used combined to express more conventional characteristics. The eigen-characteristics are the following: concurrency, transaction length, contention, working-set size, pollution, temporal locality, predominance and density. We have evaluated three of them:

- *Concurrency*: It defines the number of concurrently running threads of the application.
- *Transaction Length*: Defined as the number of reads and writes inside a transaction, it can be worked out by adding `R_HOT`, `W_HOT`, `R_MILD` and `W_MILD`.
- *Contention*: The probability of conflict of a transac-

tion. See Section IV-B.

Spatial locality is added to the eigen-characteristics above, and its effect is also discussed.

IV. Scalability Analysis

A. Methodology

We use Simics [16], a full system execution-driven simulator, to make the scalability analysis. Simics simulates the Sun Fire server brand and the SPARC architecture and it is able to run an unmodified copy of a Solaris operating system. Solaris 10 was installed in the simulated machine.

A 16-core CMP system was considered for simulation. Each in-order single-issue core has a 32KB, 4-way, 64B block private L1 I and D cache. L2 cache is unified and shared, with a capacity of 8MB organized in 16 banks, 8 ways and 64B blocks. Cache coherence is based on the MESI protocol with an on-chip directory holding a bit vector of sharers per block.

As regards the TM system, we use the GEMS module [17], which is provided by the Wisconsin Multifacet Project as an open-source module for Simics. GEMS's Ruby module implements the LogTM-SE HTM [5] and also includes a detailed timing model for the memory system. Ruby was modified to include all enhanced signature schemes analyzed in this work: $(3, \delta_P)$-LS-Sig, $(5, \delta_P)$-LS-Sig, MS $s = 3$ L2 Sig, ASYM $a = 6$ and $a = 5$ signatures. We compare their performance with that of conventional parallel signatures, and perfect signatures. The latter do not yield false positives. We set the signature size to 8Kbit, 4Kbit for the read set and 4Kbit for the write set[1]. All filters used 4 hash functions of the H3 family and the same H3 matrices of Ruby. We used 15 out of 16 cores in the simulated system. The remaining processor is left to the OS so that it does not interrupt simulations.

Finally, Ruby adds pseudorandom delays to the latency of memory accesses to deal with variability in simulation experiments. Therefore, multiple runs of each experiment were done to obtain confident error bars [18].

B. Contention Results

Contention is defined in [15] as the probability of conflict of a transaction, and an expected value is proposed:

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

Expression 1 is deduced as follows. Let W'_{HOT} and R'_{HOT} be the number of accesses to different addresses in the hot array. R'_{HOT} can be defined as

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

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

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

Figure 1 shows the results obtained from the simulator for the contention experiments. The first graph depicts the results of the parameter configuration that defines short transactions. We can see that the unprotected version of the code do not achieve the maximum speedup available, which is 15. Instead, it is $11\times$ as fast as the serial version. The problem lies in the implementation of EigenBench, as the mild array is of size $N_{MILD} * N_{TH}$, so the serial version works with a mild array of size N_{MILD}, whereas in the concurrent version, the mild array is 15 times larger. Then, the cache hierarchy makes the serial version goes faster than the parallel one. Also, the network traffic increases since the hot array is shared between 15 cores, and this gets worse as contention is higher. The graph also shows that all signature variants perform the same as perfect signatures when transactions are short and signature size is large enough (8Kbit). Notice that enhanced schemes do not harm the performance of short transactions.

Next experiments use long transactions, and signature length is kept at 4Kbit per data set. Table III shows the parameters we used. R_{MILD} and W_{MILD} changed to 200. The rest of parameters are the same except for R_{OUT} and W_{OUT} that were modified to maintain 80% of predominance. The second graph in Figure 1 shows the results obtained without spatial locality. The speedup now drops significantly due to the aforementioned issues. However, enhanced signatures perform better than the parallel signature version since, although the accesses are randomly distributed, some of them are arbitrarily nearby enough to take advantage of locality-sensitive signatures. The remaining plots in Figure 1 show results in which spatial locality is set to 25%, 50% and 75% respectively. The speedup of all versions improves as locality increases, as the cache hierarchy is better harnessed. Also, the system works at 64B block granularity, so certain accesses nearby each other will be in the same cache memory block as the arrays comprise *long* elements of 4B each. Thereby, spatial locality implies some amount of temporal locality which improves the performance. In any case, enhanced signature proposals perform similar to or better than the parallel signature in the explored cases, and they practically match the performance of perfect signatures when locality is 75% in the last graph of Figure 1. Finally, note that too much contention can lead the HTM system to perform worse than the serial version.

C. Concurrency Results

In this section we study the scalability of signatures in terms of concurrency. The parameters in the "Concurrency"

[1]Sanchez et al. [6] perform a study of signatures in real systems, where they use such sizing values.

TABLE III: Parameters of EigenBench for the experiments.

	Contention		Concurrency		Xact Length			
Param	Xact Length: Short	Xact Length: Long	Non-local	Local	Sym	Asym	Transactional Parallel Input	Unprotected Serial Input
N_{TH}	15	15	[1, 15]	[1, 15]	15	15	15	1
$loops$	128	128	[1920, 128]	[1920, 128]	128	128	128	1920
N_{HOT}	$[1K, 128K]$	$[1K, 128K]$	128K	128K	128K	128K	0	0
N_{MILD}	1M	1M	1M	1M	1M	1M	8M	120M
R_{HOT}	45	45	45	45	0	0	0	0
W_{HOT}	5	5	5	5	0	0	0	0
R_{MILD}	45	200	200	200	[20, 320]	[30, 480]	[10, 100]	[10, 100]
W_{MILD}	5	200	200	200	[20, 320]	[10, 160]	[10, 100]	[10, 100]
R_{OUT}	18	49	49	49	0	0	0	0
W_{OUT}	2	41	41	41	0	0	0	0
lct	0	0	0	0	0	0	0	0
lcs	0	0, 0.25, 0.5, 0.75	0	0.5	0.25	0.25	0, 0.25, 0.5, 0.75	0, 0.25, 0.5, 0.75

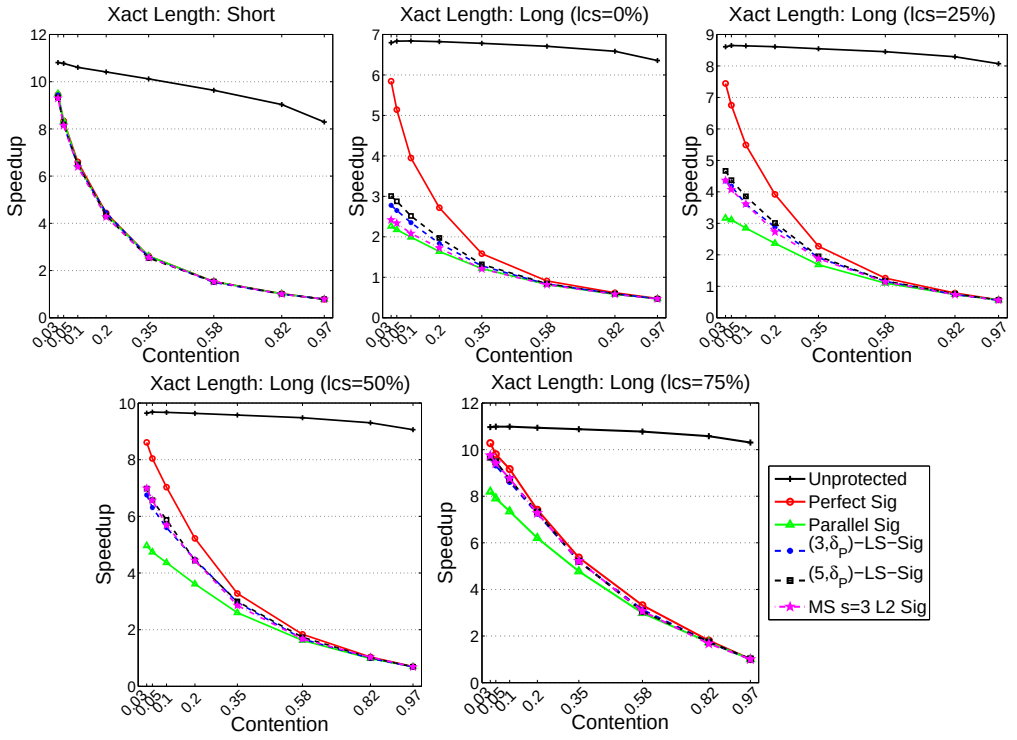

Fig. 1: Contention results for 15 threads.

columns of Table III were used to perform the experiments, and they define large transactions of 550 accesses spread out between reads and writes, with a small fraction over the hot array. These parameters yield a 3% of real contention, and the expected contention due to signature false positives should be lower, since we are dealing with 8Kbit signatures and transactions of 550 accesses. From the equation of false positive probability for a Bloom filter [19]:

$$p_{\text{FP}}(M, n, k) = \left(1 - \left(1 - \frac{1}{M}\right)^{nk}\right)^{k} \quad (2)$$

where M is the signature size, n the number of insertions and k the number of hash functions, we get about 0.2% false positives per filter.

Left graph in Figure 2 shows the results obtained in the absence of spatial locality. The speedup of Unprotected do not get to 15 because the mild array is of size $N_{MILD} * N_{TH}$, so the serial version works with a mild array of size N_{MILD} that is better managed by the cache hierarchy. We have measured an increasing speedup of $(5, \delta_P)$-LS-Sig with respect to the conventional parallel signature of 1, 1.02, 1.08, 1.23 and 1.33, for 1, 2, 4, 8 and 15 threads respectively. As for the false positive percentage, we have measured the following values for 1 to 15 threads: (0.04, 0.02), (0.6, 0.6), (2.0, 1.7), (3.6, 2.8) and (5.1, 3.5). These values are pairs (RS filter, WS filter) for the percentage of false positives of the conventional parallel signature. We can see that percentages are low when concurrency is low, because there are few checks to the signatures. However, as concurrency increases, the percentage of false positives rises promptly. For $(5, \delta_P)$-LS-Sig the false positive percentages are lower: (0.06, 0.04), (0.5, 0.4), (1.6, 1.2), (3.0, 2.0) and (4.3, 2.7).

The results depicted in the right graph of Figure 2 include

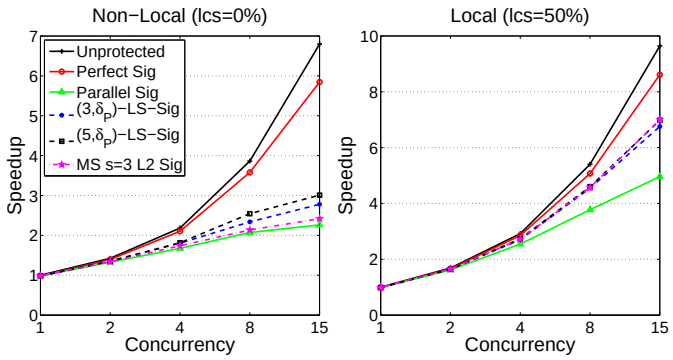

Fig. 2: Concurrency results for 15 threads.

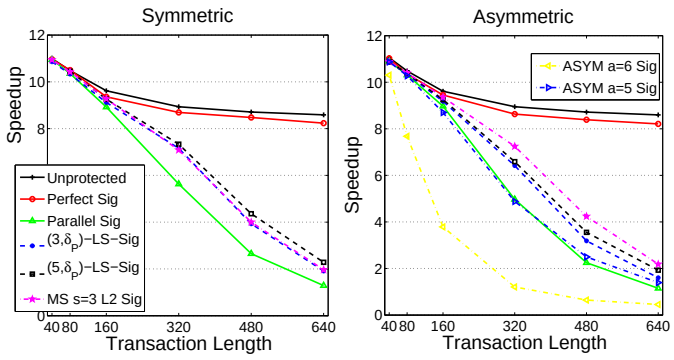

Fig. 3: Transaction length results for 15 threads.

the locality parameter set to 50%. Now, we obtain better results when comparing $(5, \delta_P)$-LS-Sig to parallel conventional signatures. In this case, we get a relative speedup of 1, 1.02, 1.08, 1.23 and 1.41. We can see that the enhanced signature proposals scale better than conventional parallel signatures. However, the more cores are available, the worse the effect of false conflicts with imperfect signatures, as there are more signature checks and the probability of getting a false positive increases. So, even with enhanced signatures, we have to keep the rate of false positives low.

D. Transaction Length Results

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

Figure 3, on the left, shows the results for the symmetric configuration parameters in Table III. Maximum speedup is about $11\times$ the serial for the shortest transaction length of 40 elements. As we discussed in the last section, this is due to the working set effect. Perfect signatures perform similar to the unprotected version in this case, as contention is set to 0. However, a small performance drop can be appreciated as transaction length increases, due to a small fraction of aborts that are caused by false sharing. The results for imperfect signatures, the parallel and the enhanced ones, get affected by false conflicts due to false positives in the filters. The performance with parallel signatures drops quickly from transaction length 160 onwards. Parallel signatures of 4Kbit per set match the performance of the serial version for transaction length 640. However, the enhanced signature schemes perform better than parallel signatures, although they exhibit a considerable performance degradation with respect to perfect signatures from transaction length 480 onwards.

The right graph in Figure 3 depicts the results obtained for different transaction lengths and asymmetric data sets. Now, the read set is three times as large as the write set, as seen in Table III. We can see that the best results using imperfect signatures are yielded by MS $s = 3$ L2 signatures, that can cope with the data set asymmetry. Reconfigurable asymmetric

TABLE IV: RS and WS lengths measured by the HTM system compared to that of EigenBench input parameters. Transaction length $|RS| + |WS| = |DS|$, and RS to WS ratio $\frac{|RS|}{|WS|}$.

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

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

Next, we conduct a batch of experiments to see the relationship between transaction length and signature size, and we show their iso-speedup curves. We modified EigenBench to escape implicit accesses that should not be tracked by the HTM system, as said in Section III-A. Table III summarizes the input parameters used for the experiments. Every transactional parallel workload is compared to its corresponding unprotected serial version, whose input parameters are shown in the column "Unprotected Serial Input". Notice that the size of the mild array, N_{MILD}, is 8M as each of the 15 threads has its private subarray. Also, the number of loops is divided by 15. Read and write sets are symmetric and range from 10 to 100 step

TABLE V: Number of transactional accesses issued by EigenBench versus number of blocks where they are mapped depending on the amount of locality.

EigenBench |RS|+|WS|	TM avg |RS|+|WS| (blocks)			
	lcs=0%	lcs=25%	lcs=50%	lcs=75%
20	20	19	17	13
40	40	37	32	25
60	60	55	47	36
80	80	73	62	47
100	100	91	78	58
120	120	109	93	69
140	140	127	108	80
160	160	145	123	92
180	180	163	139	103
200	200	180	154	114

10. Locality varies from 0 to 0.75.

Figure 5 shows the speedup of conventional parallel signatures as we vary signature size and transaction length. We can see that using 8K signatures the speedup is maximum, although Figure 3 shows that increasing transaction size will end up degrading the performance of such a signature. The variability introduced by the simulator is the cause for those irregular iso-speedup curves for 14.5 and 14.8. On the other hand, small signatures of 256 or 512 bits and large transactions, greater than 70, can lead to such a degradation that the transactional system is worse than executing the serial version of the program. We can see speedups of 0.75 and below.

Spatial locality of reference varies from 0% to 75% and, as locality increases curves shift to the left to such an extent that we get an speedup of 14 with 256bit signatures, transactions of length 20 and 75% of locality, instead of getting a speedup of 11 in the absence of locality. The reason behind this behavior is that the simulator implements conflict detection at cache block granularity, specifically, 64B blocks are used, whereas EigenBench issues 4 byte variable accesses. Then, when there is no locality, each 4 byte variable is mapped to one block only. However, when locality is introduced by incrementing the `lcs` parameter, several variables happen to be in the same block. Table V shows the number of variables issued by EigenBench and the actual measured number of blocks which are tracked by signatures system. We can see that the greater the locality the lesser number of blocks and, consequently, the occupancy of the filters is reduced and so is the number of false positives.

As far as LS-Sig is concerned, Figure 6 shows iso-speedup plots for $(5, \delta_P)$-LS-Sig with `lcs` varying from 25% to 75%. Now, besides the benefit from locality per se that can be seen in Figure 5, the LS-Sig is able to extract even more performance from such a feature. The more locality the more speedup. Figure 4 shows the relative speedup of LS-Sig over conventional parallel signatures. The graph corresponds to `lcs`=75%, where the maximum speedup gets to 98%.

V. Conclusions

This work discusses the scalability of state-of-the-art signatures for transactional memory systems. We study their behavior when stressing different application characteristics like contention, concurrency, transaction length and spatial locality. To that end, we modify and use the Stanford's EigenBench benchmark that allows the emulation of a set of orthogonal application characteristics.

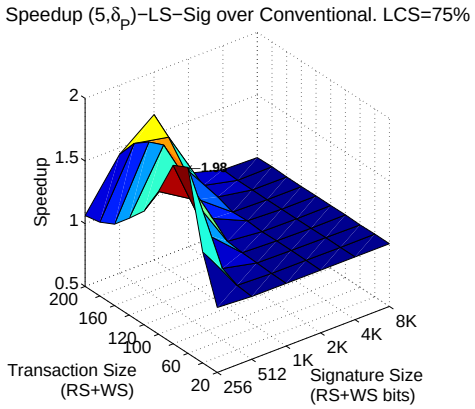

Fig. 4: Speedup of $(5, \delta_P)$-LS-Sig over conventional parallel signatures with lcs equal 75%. Maximum speedup is shown.

Results show that block granularity conflict detection significantly enhance performance of signatures in the absence of real conflicts. However, experiments must be carried out in order to determine if false conflicts due to block false sharing can degrade performance to such an extent that the benefit from block granularity conflict detection is lost. On the other hand, enhanced signatures (LS, MS) improve the overall execution and could be thought of a way of getting the same performance than conventional signatures while either working at word granularity or reducing hardware requirements.

Another implication that can be drawn from the experiments is that large transactions or small signatures can lead the TM system to perform worse than serial. For conventional parallel signatures, such a degradation can happen when transactions are about one fifth the signature size. With enhanced signatures, the filter can reach higher occupancy without degradation, but it depends on the amount of locality. Unfortunately, both conventional and enhanced signatures converge when transactions are very large. Not to mention that increasing concurrency would increase signature checks and the probability of false positive as well. A system to stop insertions into filters that have gone beyond a given threshold should be explored to try to avoid this situation.

Results also suggest that implicit transactional memory systems, although easier to program, can harm the performance of signatures since more transactional data have to be tracked. Finally, having large filters could be a waste of power when we have small transactions. A scalable Bloom filter is worth exploring for these cases.

References

[1] M. Herlihy and J. Moss, "Transactional memory: Architectural support for lock-free data structures," in *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ser. ISCA'93, 1993, pp. 289–300.

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

[3] L. Ceze, J. Tuck, C. Cascaval, and J. Torrellas, "Bulk disambiguation of speculative threads in multiprocessors," in *Proceedings of the 33th Annual International Symposium on Computer Architecture*, ser. ISCA'06, 2006, pp. 227–238.

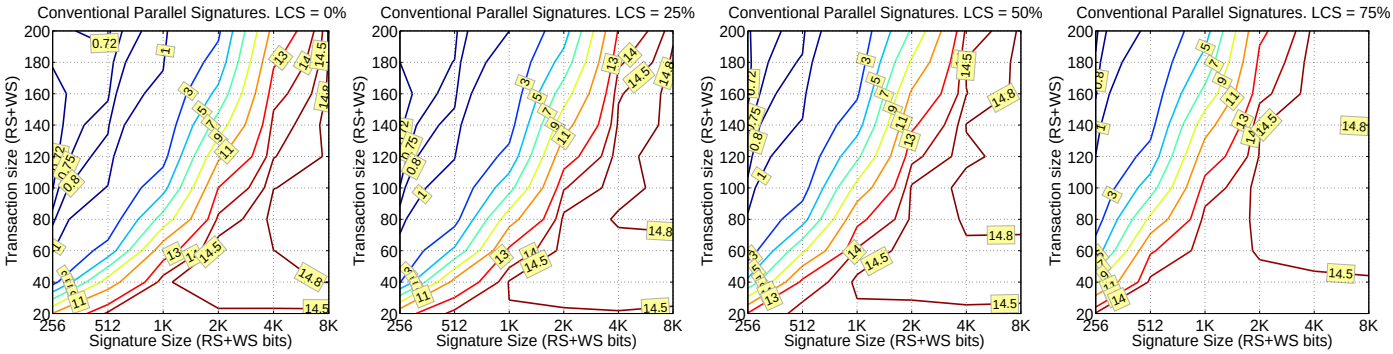

Fig. 5: Iso-speedup contour plots of conventional parallel signatures with lcs ranging from 0% (left) to 75% (right).

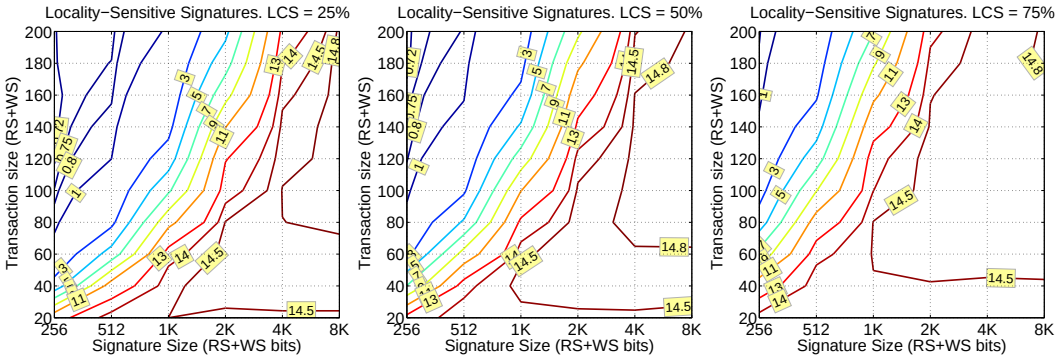

Fig. 6: Iso-speedup contour plots of $(5, \delta_P)$-LS-Sig with lcs ranging from 25% (left) to 75% (right).

[4] M. Mehrara, J. Hao, P. Hsu, and S. Mahlke, "Parallelizing sequential applications on commodity hardware using a low-cost software transactional memory," in *Proceedings of the ACM SIGPLAN Conference on Programming Languages Design and Implementation*, ser. PLDI'09, 2009, pp. 166–176.

[5] 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 *Proceedings of the 13th International Symposium on High-Performance Computer Architecture*, ser. HPCA'07, 2007, pp. 261–272.

[6] D. Sanchez, L. Yen, M. Hill, and K. Sankaralingam, "Implementing signatures for transactional memory," in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO'07, 2007, pp. 123–133.

[7] R. Quislant, E. Gutierrez, O. Plata, and E. L. Zapata, "LS-Sig: Locality-sensitive signatures for transactional memory," *IEEE Trans. on Computers*, vol. 62, no. 2, pp. 322–335, 2013.

[8] R. Quislant, E. Gutierrez, O. Plata, and E. Zapata, "Hardware signature designs to deal with asymmetry in transactional data sets," *IEEE Trans. on Parallel and Distributed Systems*, vol. 24, no. 3, pp. 506–519, 2013.

[9] W. Choi and J. Draper, "Unified signatures for improving performance in transactional memory," in *IEEE Int'l. Parallel Distributed Processing Symp. (IPDPS'11)*, May 2011, pp. 817–827.

[10] D. Dice, O. Shalev, and N. Shavit, "Transactional locking ii," in *Proceedings of the 20th International Symposium on Distributed Computing*, ser. (DISC'06), 2006, pp. 194–208.

[11] C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun, "STAMP: Stanford Transactional Applications for Multi-Processing," in *Proceedings of the IEEE International Symposium on Workload Characterization*, ser. IISWC'08, 2008, pp. 35–46.

[12] M. Moravan, J. Bobba, K. Moore, L. Yen, M. Hill, B. Liblit, M. Swift, and D. Wood, "Supporting nested transactional memory in LogTM," in *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, ser. ASPLOS-XII, 2006, pp. 359–370.

[13] D. Thiebaut, J. Wolf, and H. Stone, "Synthetic traces for trace-driven simulation of cache memories," *IEEE Transactions on Computers*, vol. 41, pp. 388–410, 1992.

[14] 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.

[15] S. Hong, T. Oguntebi, J. Casper, N. Bronson, C. Kozyrakis, and K. Olukotun, "Eigenbench: A simple exploration tool for orthogonal TM characteristics," in *Proceedings of the IEEE International Symposium on Workload Characterization*, ser. IISWC'10, 2010, pp. 1–11.

[16] 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.

[17] 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 Computer Architecture News*, vol. 33, no. 4, pp. 92–99, 2005.

[18] A. Alameldeen and D. Wood, "Variability in architectural simulations of multi-threaded workloads," in *Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, ser. HPCA'03, 2003, pp. 7–18.

[19] P. Bose, H. Guo, E. Kranakis, A. Maheshwari, P. Morin, J. Morrison, M. Smid, and Y. Tang, "On the false-positive rate of Bloom filters," *Information Processing Letters*, vol. 108, no. 4, pp. 210–213, 2008.