Appears in the Proceedings of the
40th Annual IEEE/ACM Symposium on Microarchitecture (MICRO-40), 2007

Implementing Signatures for Transactional Memory

Daniel Sanchez, Luke Yen, Mark D. Hill, Karthikeyan Sankaralingam
Department of Computer Sciences, University of Wisconsin-Madison
http://www.cs.wisc.edu/multifacet/logtm
$\{$*daniel, lyen, markhill, karu*$\}$*@cs.wisc.edu*

Abstract

Transactional Memory (TM) *systems must track the read and write sets—items read and written during a transaction—to detect conflicts among concurrent transactions. Several TMs use* signatures, *which summarize unbounded read/write sets in bounded hardware at a performance cost of* false positives *(conflicts detected when none exists).*

This paper examines different organizations to achieve hardware-efficient and accurate TM signatures. First, we find that implementing each signature with a single k-hash-function Bloom filter (True Bloom signature) *is inefficient, as it requires multi-ported SRAMs. Instead, we advocate using k single-hash-function Bloom filters in parallel* (Parallel Bloom signature)*, using area-efficient single-ported SRAMs. Our formal analysis shows that both organizations perform equally well in theory and our simulation-based evaluation shows this to hold approximately in practice. We also show that by choosing high-quality hash functions we can achieve signature designs noticeably more accurate than the previously proposed implementations. Finally, we adapt Pagh and Rodler's* cuckoo hashing *to implement* Cuckoo-Bloom signatures. *While this representation does not support set intersection, it mitigates false positives for the common case of small read/write sets and performs like a Bloom filter for large sets.*

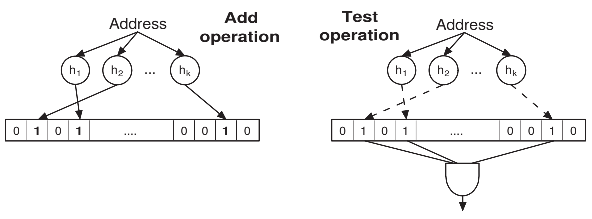
1. Introduction

Transactional memory (TM) [13, 15] systems ease multithreaded programming by guaranteeing that some dynamic code sequences, called *transactions*, execute atomically and in isolation. To achieve high performance, TMs execute multiple transactions concurrently and commit only those that do not conflict. A *conflict* occurs when two concurrent transactions perform an access to the same memory address and at least one of the accesses is a write. A TM system must implement mechanisms to detect these events (conflict detection).

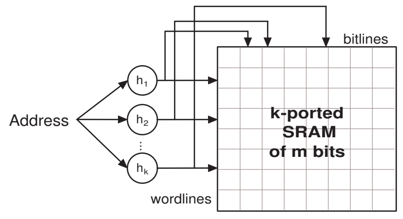
An important aspect of conflict detection is recording the addresses that a transaction reads (*read set*) and writes (*write set*) at some granularity (e.g., memory block or word). One promising approach is to use *signatures*, data structures that can represent an unbounded number of elements *approximately* in a bounded amount of state. Led by Bulk [7], several systems including LogTM-SE [31], BulkSC [8], and SigTM [17], have implemented read/write sets with per-thread hardware signatures built with Bloom filters [2]. These systems track the addresses read/written in a transaction by inserting them into the read/write signatures, and clear both signatures as the transaction commits or aborts. Depending on the system, signatures must also support testing whether an address is represented in it or intersecting two signatures. A test or intersection operation may signal a conflict when none existed (a *false positive*), but may not miss a conflict (a *false negative*). False positives cause unnecessary conflicts that may degrade performance, but they do not violate transaction atomicity.

This paper seeks to improve the performance and to reduce the cost of hardware signature implementations. The three main functional requirements of signature implementations are: (a) they should minimize gratuitous aliases for small read/write sets, (b) they should gracefully degrade as read/write sets become large, and (c) performance should be robust to changes in workload and system size. Hardware signature implementations should be cost-efficient (e.g., by efficiently using state and cheaply implementing state and logic). This paper explores the design space of signatures with formal analysis, area analysis, and experimental performance evaluation of signatures. Our contributions include:

- We show that *true Bloom signatures*, implemented with a single Bloom filter of k hash functions and m state bits, are area inefficient when implemented as k-ported memories.
- Rather we advocate *parallel Bloom signatures* that use k parallel Bloom filters, each with one hash function and m/k state bits, and only require single-ported memories. We show with probabilistic analysis that paral-



(a) Design



(b) Hardware implementation

Figure 1: True Bloom Signatures

lel Bloom signatures perform asymptotically the same as true Bloom signatures when $k/m \ll 1$.

- We advocate using high-quality hash functions (e.g. H_3 [5]) to achieve more accurate signatures with practical address streams, and our results show that these hash functions achieve better results in practice than those used in previous designs.
- We present a different signature implementation that can out-perform Bloom signatures. Our Cuckoo-Bloom signature adapts Cuckoo hashing [20] to represent addresses in a way similar to a cuckoo hash table for small sets and morphs to a Bloom filter as the hash table fills up. However, Cuckoo-Bloom signatures add complexity and do not support signature intersection.
- Finally, we examine the performance of signature implementations in the context of LogTM-SE [31], which tests signature membership on coherence events. The main conclusions from our simulation-based performance study is that parallel Bloom signatures match the performance of true Bloom signatures and that using an L2 cache directory to filter signature checks can mitigate false positives as the number of cores increases.

We next present the design, analysis and implementation of true Bloom signatures (Section 2), parallel Bloom signatures (Section 3), and our newly-developed Cuckoo-Bloom signatures (Section 4). We then examine area requirements (Section 5), evaluate performance (Section 6), review related work (Section 7), and conclude the paper (Section 8).

2. True Bloom signatures

All currently proposed TM systems that use hardware signatures advocate implementations using Bloom filters [2]. We call a signature implemented with a *single* Bloom filter a *true Bloom signature*. In this section we review true Bloom signatures, analyze their false positive rates, and sketch a hardware implementation using multi-ported SRAMs.

Design: A true Bloom signature provides an efficient way to represent a set of values (in our case, block addresses). Inserting new addresses and testing for membership in the signature is simple, with a certain probability of false positives and no possibility of false negatives. A true Bloom signature consists of an m-bit field, which is accessed using k independent hash functions, as shown in Figure 1a.

Every bit in the field is initially set to 0. To insert an address to the set, the k hash values of the address are computed. Each hash function h_i can give a value in the range $[0, ..., m-1]$. The bits at the positions indicated by these values are set to 1. To test for membership of an address, we compute the results of the hash functions and check the contents of the bits they point to. If at least one bit is set to 0, the address is not in the signature. If all the bits are set to 1, either the address is in the set or the insertion of other addresses set these bits to 1 (a false positive).

Analysis: We now present a formal analysis of false positives, which are critical to performance. Let us assume that we insert n addresses to the filter, and that the k-tuples of hash values are independent and uniformly distributed. This is approximately the case even with practical address streams if we use universal or almost-universal hash functions [23]. On a single insertion, the probability of a particular hash function writing a 1 to the i-th position on the bit array (regardless of whether this position was 0 or 1) is $1/m$. Since the k hash functions are independent, the probability of not setting a certain bit to 1 in one insertion operation is $(1-1/m)^k$. Therefore, the probability of a single bit still being 0 after the n insertions is $p_0(n) = (1-1/m)^{nk}$.

On a test for membership, the test returns true only if all of the checked bits are set to one. The probability of getting a positive match is:

$$P_P(n) = (1 - p_0(n))^k$$

However, we are interested in the probability of *false* positives, i.e. the probability that a hit occurs *and* that the address we test for is not one of the n inserted addresses. In general, this is:

$$P_{FP}(n) = P_P(n) \times P_I(n)$$

where $P_I(n)$, by Bayes' rule, is the probability that the address was not inserted into the signature, conditioned by a positive test result in the signature. Assuming that the addresses we test for are independent from the inserted ones, uniformly distributed, and that the number of addresses n_t we test for is much larger than the set of inserted addresses (i.e. $n_t \gg n$), then $P_I(n) = \frac{n_t - n}{n_t} \cong 1$ and:

$$P_{FP}(n) \cong P_P(n) = (1 - p_0(n))^k$$

Experimentally, however, the probability of a false positive may differ from the probability of a positive, because addresses are not random (e.g. locality) and hash functions might not be perfectly universal [22, 23]. Nevertheless, we find this equation to be a good first-order approximation, and at the very least it is an upper bound on the probability of false positives (because $P_I(n) \leq 1$).

Finally, let us perform an approximation to simplify the equation of $P_{FP}(n)$, which will also be useful in the next section. Consider the Taylor series expansion of the exponential function, $e^x = \sum_{n=0}^{\infty} \frac{1}{n!} x^n$. For our purposes, the number of bits m is relatively large, so $|1/m| \ll 1$ and:

$$e^{-1/m} = 1 - \frac{1}{m} + \frac{1}{2m^2} - \frac{1}{6m^3} + ... \cong 1 - \frac{1}{m}$$

Therefore, $p_0(n) = (1 - 1/m)^{nk} \cong e^{-\frac{nk}{m}}$, and:

$$P_{FP}(n) \cong \left(1 - e^{-\frac{nk}{m}}\right)^k \text{ when } \frac{1}{m} \ll 1$$

Design dimensions: There are three *design dimensions* in a true Bloom signature: (a) the size of the bit field, (b) the number of hash functions, and (c) the hash functions themselves. The size of the bit field (m) is a critical parameter: a larger field decreases the probability of false positives for a certain number of insertions ($P_{FP}(n)$), but it increases the hardware requirements as well.

The effect of the number of hash functions is depicted in Figure 2, which shows the probability of false positives in 1024-bit true Bloom signatures as we vary the number of addresses inserted, n, on the x-axis, and the number of hash functions, k (different lines). In the main figure, n varies from 0 to 1000, while the close-up focuses on n up to 120. For example, when $n = 20$ addresses have been inserted, a filter with $k = 1$ has $P_{FP}(n) = 0.02$, and a filter with $k = 4$ has $P_{FP}(n) = 3 \times 10^{-5}$. For $n = 600$ addresses, the probabilities are now 0.44 and 0.67, and the one hash function filter outperforms the four hash function filter. More generally, increasing the number of hash functions (larger k), reduces false positives for small read/write sets (an important case) at the cost of more false positives for large read/write sets.

Finally, signatures need to implement k different hash functions. Previously proposed TM signatures use bit-selection, where each hash value comes from a subset of the bits of the address. While bit-selection is simple, it may not yield sufficient variation to approximate a universal hash function. We advocate using functions from the H_3 family of universal hash functions [5, 23], which can achieve many uncorrelated and uniformly distributed hash values. In an H_3 hash function, each bit of each hash value is generated by XORing a subset of the bits of the address.

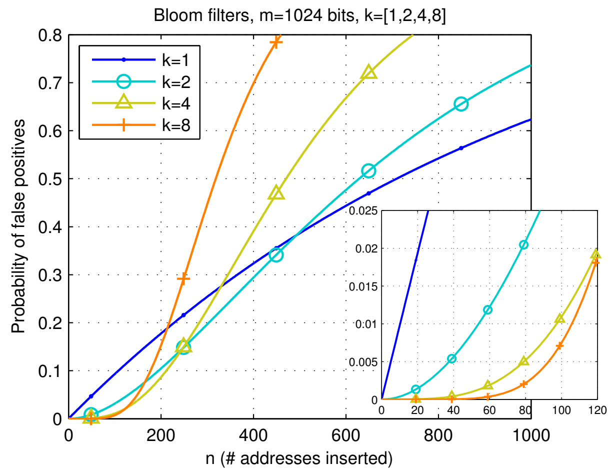


Figure 2: Influence of the number of hash functions on the probability of false positives

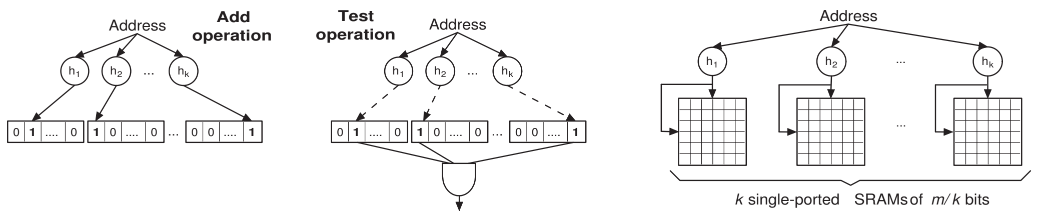
These address bits are randomly chosen, with the probability of selecting each individual address bit being 0.5. In this paper we closely follow the definition of the H_3 hash functions, but actual implementations could explore XORing fewer address bits to reduce hardware cost [28].

Hardware: True Bloom signatures can be implemented in hardware by partitioning the bit-field into words, and storing them in a small, bit-addressable SRAM. As shown in Figure 1b, we divide the output bits of the generated hash values into wordlines and bitlines that address the SRAM. To insert an address, for each hash value, the appropriate wordline is raised and the corresponding bitline is driven to high, while the other bitlines are left floating. To test for an address, the bit addressed by each hash value is read by raising the appropriate wordline and sensing the bitline's value.

Regarding the hash functions, bit-selection requires trivial hardware, and hardwired H_3 hash functions are relatively inexpensive to implement, requiring a small tree of 2-input XOR gates per bit of each hash function. To implement k hash function signatures, we should use SRAMs with k read and write ports (we could still use a single-ported SRAM and perform the reads or writes over multiple cycles, but that would complicate the control logic and increase the delay). This is not area-efficient for filters with multiple hash functions, because the size of an SRAM cell increases quadratically with the number of ports. In the next section, we describe a partitioning strategy to overcome this quadratic growth.

3. Parallel Bloom signatures

This section describes *parallel Bloom signatures*, which, we will show, perform like true Bloom signatures, but avoid multi-ported SRAMs.



(a) Design (b) Hardware implementation

Figure 3: Parallel Bloom Signatures

Design: Instead of having a single k-hash function Bloom filter, we now consider using k Bloom filters, each with a different hash function. To have the same amount of state as a true Bloom signature, each of the individual Bloom filters uses a m/k-bit field. To insert an address, we hash it and set a bit in all k filters. We report that an address is represented in the signature only if all the individual Bloom filters say so. We call this structure a *parallel Bloom signature* and its design is shown in Figure 3a. A similar design was proposed in Bulk [7].

Analysis: Under the same assumptions made for the analysis of true Bloom signatures, the probability of the i-th bit of a particular filter being still 0 after n insertions is $p_0(n) = \left(1 - \frac{1}{m/k}\right)^n$. Hence, the probability of false positives is:

$$P_{FP}(n) \cong P_P(n) = \left(1 - \left(1 - \frac{1}{m/k}\right)^n\right)^k$$

This appears to be different from the $P_{FP}(n)$ for true Bloom signatures. However, if we apply the Taylor series approximation of e^x as we did before, we obtain:

$$\frac{k}{m} \ll 1 \Rightarrow 1 - \frac{1}{m/k} \cong e^{-\frac{k}{m}}$$

And therefore,

$$P_{FP}(n) \cong \left(1 - e^{-\frac{nk}{m}}\right)^k \text{ when } \frac{k}{m} \ll 1$$

Under the approximation, *a parallel Bloom signature achieves the same $P_{FP}(n)$ as a true Bloom signature*. This approximation is very accurate when the number of hash functions is much smaller than the length of the bit field (i.e. $k/m \ll 1$), which will normally be the case. Moreover, we experimentally verify this result in Section 6. A similar approximation is also used in filters in networking [9], but without realizing that one hash function per parallel filter is sufficient and can lead to a more area-efficient design.

Hardware: The implication of such a partitioning for hardware is that instead of implementing multi-hash function Bloom filters with multi-ported large SRAMs, we can use multiple, smaller single-ported SRAMs. Finally, the hash functions are also less expensive to implement, because they now generate hash values that are smaller by a factor of k. Figure 3b shows a canonical hardware implementation.

4. Cuckoo-Bloom signatures

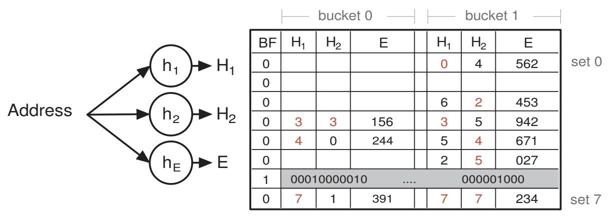
So far, we have considered signatures implemented with Bloom filters only. In this section, we present a new signature implementation, called a *Cuckoo-Bloom signature*.

Cuckoo-Bloom signatures represent small read/write sets by adapting cuckoo hashing [20]. We choose a hash table based scheme because similar approximate membership testers have proven more space-efficient than Bloom filters [6, 19]. Cuckoo hashing supports a fast lookup via two (parallel) probes, but complicates inserts. When an insert finds both probe targets full, it removes one of the old target items, inserts the new item, re-inserts the old at its other probe target, and recursively repeats. This allows us to reach high occupancies, in the range of 80%. Optimizing for fast signature lookup makes sense, because LogTM-SE experiments show lookups can be 5.5 to 200 times more frequent than inserts. However, a hash table alone can only hold a limited number of entries. Because we want to represent an unbounded number of addresses, we dynamically transform the hash table into a Bloom filter as occupancy increases.

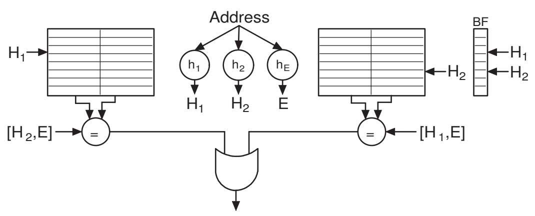
Cuckoo-Bloom signatures match the low false positive rates of Bloom signatures with many hash functions when the number of addresses is small (an important and common case), and show the good asymptotic behavior of Bloom signatures with few hash functions when the number of addresses is large. We now present the design, analysis, and hardware implementation of Cuckoo-Bloom signatures.

Design: The basic structure of the signature is shown in Figure 4. The table has S sets (rows), and each of those sets is divided in B buckets, like a B-ary set-associative cache. Each of the cells in the table stores a 3-tuple of hash values of one address.

On an insert operation, three hash functions (h_1, h_2, h_E) are applied to the address, yielding the hash values (H_1, H_2, E). H_1 and H_2 are used to index the table, while E provides extra information about the address that makes its representation more accurate. In the simplest case, at least one bucket of the two indexed sets will be unused. In that case, the two sets are retrieved, and the new element is inserted in the last bucket of the least occu-



(a) Design



(b) Hardware implementation

Figure 4: Cuckoo-Bloom Signatures

pied set.

To test if an address was inserted into the structure, we compute the (H_1, H_2, E) tuple, retrieve the sets addressed by H_1 and H_2, and check if an element with (H_1, H_2, E) is already present.

If, when doing an insertion, both sets happen to be full, the element in the leftmost bucket of one of those two sets is evicted, the remaining buckets of the set are shifted to the left, and the new element is inserted into the last bucket. We then re-insert this evicted element back into the table, possibly evicting another one, and repeat the process as needed. For example, if we tried to insert $(3, 3, 54)$ into the structure in Figure 4a, $(3, 3, 156)$ would be evicted, $(3, 5, 942)$ would be shifted into bucket 0, and $(3, 3, 54)$ would be inserted into bucket 1 of set 3. The evicted element $(3, 3, 156)$ would then be re-inserted into bucket 1 of set 3, evicting $(3, 5, 942)$ from bucket 0, which would in turn be inserted at bucket 1 of set 5, shifting $(2, 5, 27)$ to bucket 0 of set 5 and producing no more evictions.

As the table fills up, the insertion process could result in an infinite loop of evictions and re-insertions. To avoid this, we limit the number of iterations to a small integer (4 in our experiments). If the last iteration ends up evicting an element, we set the BF bit of one of the two possible sets of this evicted element, and convert it into a Bloom filter. This is done by first evicting all elements in the set into a separate storage space, and then hashing the new element into the Bloom filter. We then repeat the insertion process for the newly evicted elements. This chain of evictions could lead to long delays, but our analysis and experiments show that for two buckets (i.e. two elements/set) delays are acceptable, as multiple Bloom filters are rarely created. This does not hold for more buckets, where transforming the structure into a Bloom filter typically causes long delays. To achieve a low probability of false positives, each element is only hashed into one of the two sets in which it can reside. This set is determined by the least-significant bit of the E field.

Analysis: We analyzed Cuckoo-Bloom signatures using Monte Carlo simulation, as closed-form false probability formulas are difficult to obtain for the general case. Table 1 shows the parameters used for the Cuckoo-Bloom signatures and Figure 5 shows the false conflict rates. Cuckoo-Bloom signatures precisely represent the address set for small sets and morph into Bloom signatures for large sets. Cuckoo-Bloom signatures are able to perform well because we move elements around when conflicts arise, and thus achieve a high occupancy before the conversion into a Bloom filter starts (80% in our Monte Carlo simulations). Before this conversion, 95% of insertions cause no more than one eviction.

SRAM size	1024 bits	Max iterations/insert	4
Sets	32	Set size	32 bits
Buckets/set	2	Hash function length	4 bits
Hash functions	2 (H_3)	E field length	12 bits

Table 1: Parameters for Cuckoo-Bloom signatures

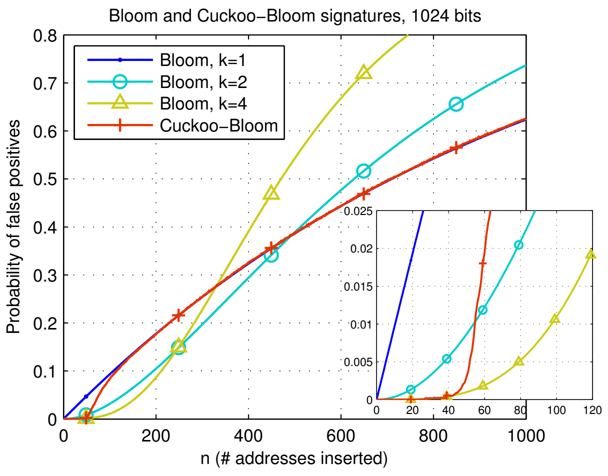


Figure 5: Probability of false positives of Bloom and Cuckoo-Bloom signatures

Hardware: Cuckoo signatures could be constructed by implementing the design in Figure 4a with a single large 2-ported SRAM. However, we can instead use two separate single-ported SRAMs. As shown in Figure 4b, each SRAM is indexed by one of the hash functions, and stores half of the sets in each table. Additionally, we just need to store one of H_1 or H_2 per entry, as the other hash value can be deduced by the set the entry is in. The BF bits are maintained as a separate array of flip-flops. We do not show the extra control logic or storage required to implement the Bloom filter upon setting the BF bit. Unlike the bit-addressed SRAMs used for Bloom signatures, writing to this Bloom filter involves reading a word, setting the corresponding bit, and writing back the entire word.

In general, this scheme can be extended to work with a higher number of buckets and multiple hash functions. Of these, all but one hash value needs to be stored in each bucket. The total number of storage bits is $S \times (B \times ((K-1) \times L_H + L_E))$, where S is the number of sets, B is the number of buckets, K is the number of hash functions (and memories), $L_H = log_2(S/K)$ is the length of the hash values, and L_E is the length of the E field. Note that the length of this field is arbitrary: a larger field will allow a more accurate representation of the address set, but more space will be required per entry.

5. Area evaluation

We have thus far analyzed the behavior and described the hardware implementation of three signature designs. In this section, we use area models derived from CACTI [27] to evaluate their area requirements. We also study the area overheads of signatures in real systems.

5.1. Area requirements of signatures

Table 2 compares the area required by true and parallel Bloom signatures for a 4Kbit signature and up to four hash functions. The area estimates were obtained using CACTI 4.2 [27], for the 65nm technology node. We used memories with 8-byte words, which yield memories of the same wordlines and bitlines for the true Bloom signature. We use dual-ended read ports, and separate read/write ports. For true Bloom signatures, we used k-ported SRAMs, and for parallel Bloom signatures we used single-ported SRAMs.

k	1	2	4
True Bloom	0.031	0.113	0.279
Parallel Bloom	0.031	0.032	0.035

Table 2: SRAM area requirements (in mm^2) of true and parallel Bloom signatures, m=4Kbit, 65nm technology

As we can see, parallel Bloom signatures use significantly less area than true Bloom signatures: 3.2× less for two hash functions, and 8× for four hash functions. While we expect a quadratic savings in area proportional to the reduction in number of ports, the savings are less due to the fixed overheads, like multiplexers or sense-amps, that these small SRAMs have. Finally, note that the partitioning strategy of parallel Bloom signatures is helpful for other implementation styles as well (e.g. for smaller signatures implemented using flip-flops, using parallel Bloom signatures greatly reduces the size of the multiplexers, decoders, and the amount of wiring).

Implementing the hash functions also contributes to the area of signatures in addition to the SRAMs. While bit-selection requires no extra logic and only wiring overhead, the H_3 hash functions require additional XOR gates to implement the hash function. Recall from Section 2 that, for n-bit addresses we need a tree of $n/2$ two-input XOR gates for each bit of the hash function. Carefully designing this hash function to use fewer address bits and physical design optimizations can reduce the area and delay of this hash function. Assuming an $n/2$ XOR tree and a 4-transistor XOR gate design [29], for $k = 4$, our area models based on transistor counts show that the hash functions occupy about one-fifth the size of the SRAM.

Finally, Cuckoo-Bloom signatures require single-ported, word-readable memories, like parallel Bloom signatures. Hence, for the same size, Cuckoo-Bloom and parallel Bloom signatures have the same memory area requirements. Cuckoo-Bloom signatures require additional BF bits, control logic for implementing evictions, and extra registers to hold evicted elements. However, these structures are likely to have a small impact on the total area and scale well with signature size.

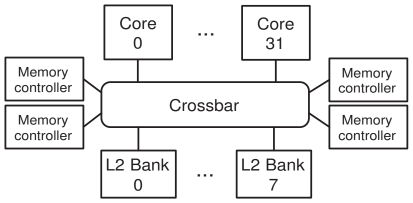
5.2. Signatures in real systems

To understand the area overheads of adding transaction support to hardware, we picked two different multi-core systems: the Sun Niagara [14], which uses simple in-order cores, and the AMD Barcelona [32], which uses more complex out-of-order cores. Although signatures are not the only extra hardware required for TM support, in this paper we focus on the area overheads of signatures alone. To make the analysis simple we picked one signature design: parallel Bloom signatures of 4Kbits with four hash functions, using the bit-selection hash functions. We assume separate signatures per thread context, and separate signatures for the read and write sets. As a result, one core of the 4-way multithreaded Niagara core will require 8 signatures. Table 4 shows the area overheads for both architectures using CACTI-based area estimates for signatures.

As we can see, the hardware required to implement signatures is noticeable in the Niagara system but minimal in the Barcelona system. The overheads differ mainly because the area for signatures scales linearly with the number of thread contexts. For the simple in-order Niagara cores, the total of 8 signatures/core amount to a total of 4Kbytes of memory, half as much as the L1 instruction cache. In terms of the overall die, the area required by signatures is at most about 1%. Also, additional signatures can be required by particular TM systems. For example, to enable virtualization, LogTM-SE [31] uses two additional signa-

Benchmark	Input	Units of work	Time in transactions	Read set size (avg/max)	Write set size (avg/max)	Dyn. instrs / xact (avg/max)	Retries/ xact
btree	Uniform random	100000 ops	54.9%	13.2 / 20	0.64 / 15	514.0 / 1988	0.022
raytrace	teapot	1 parallel phase	2.7%	5.25 / 573	1.98 / 4	11.8 / 18406	0.003
barnes	512 bodies	1 parallel phase	9.2%	5.23 / 41	3.92 / 35	200.7 / 3493	0.23
vacation	n8-q10-u80-r65536	4096 operations	100%	80.4 / 176	12.4 / 62	4301 / 318624	2.05
delaunay	gen4.2	-	100%	26.2 / 222	14.8 / 131	8331 / 89105	1.29

Table 3: Parameters and TM characteristics of the benchmarks



(a) System organization

Cores	32-way CMP, IPC/core=1, SPARCv9 ISA
L1 Caches	32KB split,4-way associative, 64B lines, 3-cycle access latency
L2 Cache	8MB, 8-way assoc., 8-banked, 64B lines, 6/20-cycle tag/data access latency
Coherence protocol	MESI, snooping, signature checks broadcast by L2
Memory subsystem	4 memory controllers, 450-cycle latency to main memory
Interconnect	Crossbar, 5-cycle link latency

(b) System parameters

Figure 6: Simulated system

	AMD Barcelona	Sun Niagara
Cores, multithreading	Quad-core, no MT	8-core, 4-way FGMT
Technology node	65nm	90nm
Die size	$291mm^2$	$379mm^2$
Core size	$28.7mm^2$	$13mm^2$
L1 areas (I/D)	$2.25mm^2$ (both)	$1.12/0.64mm^2$
Area used by signatures, per core	$0.07mm^2$	$0.54mm^2$
Core size increase	0.25%	4.1%
Die size increase	0.10%	1.1%

Table 4: Area estimates in real systems

tures per thread context (called summary signatures), hence doubling the hardware requirements. Finally, note how parallel Bloom signatures provide significant area savings over true Bloom signatures when using a large number of hash functions. If we used true Bloom signatures, we would require $4.3mm^2$ per core in the Niagara, causing a 33% increase in the core area.

Our two main conclusions from the area analysis are: (a) parallel Bloom signatures are much more area-efficient than true Bloom signatures whenever more than one hash function is used, and (b) the area required for signatures in hardware TM systems is relatively small compared to the overall processor core area.

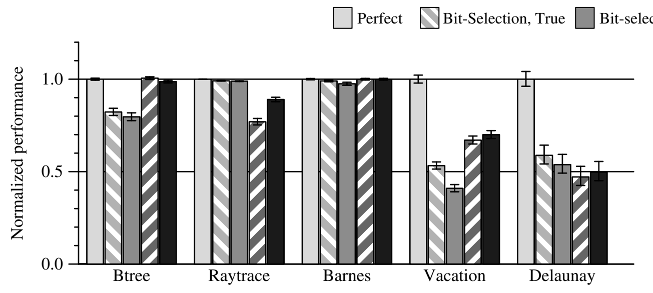
6. Performance evaluation

We now present a performance evaluation of signatures. Specifically, we evaluate true Bloom signatures, parallel Bloom signatures, and Cuckoo-Bloom signatures using the Simics full-system simulator with the GEMS [16] toolset to simulate a chip multiprocessor with LogTM-SE.

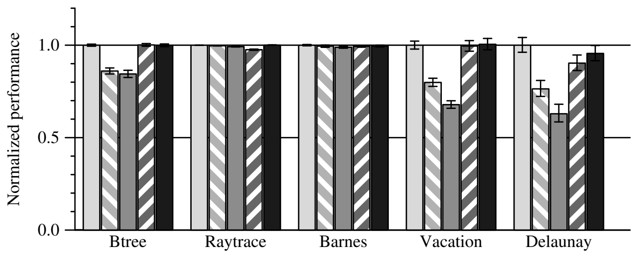
6.1. Simulation methodology

Benchmarks: We use five different benchmarks with interesting behavior relevant to the evaluation of signatures:

- **Btree:** In this microbenchmark, each thread accesses a shared B-tree to perform either a lookup or an insertion (with 80%/20% probabilities) using transactions. Per-thread memory allocators are used to increase performance.
- **Raytrace and Barnes:** Both workloads belong to the SPLASH-2 suite [30], whose transactional versions [18] feature small transactions with little contention. In this suite, Raytrace and Barnes exert the most pressure on signature designs.
- **Vacation and Delaunay:** These benchmarks belong to the STAMP benchmark suite [17]. Both feature coarse-grain, long-running transactions with large read and write sets, and follow TCC's model of all transactions, all the time [12]. Consequently, of our five benchmarks, these exert the most pressure on the signatures.



(a) 512 bits



(b) 2048 bits

Figure 7: Performance (higher is better) of true and parallel Bloom signatures with four hash functions

The exact parameters of the benchmarks, as well as their main TM-related characteristics are summarized in Table 3.

System configuration: We model a CMP modified to support LogTM-SE. The CMP has 32 in-order, single-issue cores, and has other parameters summarized in Figure 6b.

LogTM-SE is fully described elsewhere [31], but we review how it does conflict detection, as that is most pertinent to our experiments. A transactional read (write) inserts its block address into the core's read (write) signature. A transactional cache miss that is redirected to a remote core tests its address against the appropriate remote signatures (a read miss tests the write signature, while a write miss tests both signatures). A positive tests causes a negative acknowledgement that the original requesting core uses to resolve the (potentially false) conflict with a stall or trap to software (e.g., to abort).

Most of our experiments assume signature requests are broadcast to model existing broadcast proposals and put pressure on signature designs. A few experiments use LogTM-SE's original design where the L2 directory protocol filters which core sees coherence misses, and must be modified to support sticky states to handle cache victimization of transactional data [18].

Hash functions: In the evaluation of Bloom signatures, we use both bit-selection and hardwired H_3 hash functions, while using only H_3 for Cuckoo-Bloom. We use bit-interleaving as a particular class of bit-selection hash functions: of the k different hash values, the i-th hash value is generated by concatenating the bits i, $i+k$, $i+2k$, and so on. If the number of bits required for the hash functions is greater than the number of bits of the address we want to consider (25 in our case), we wrap around and start selecting bits from the beginning again. For example, the second of four 8-bit values would use address bits $(1, 5, 9, 13, 17, 21, 0, 4)$. However, we have explored other kinds of bit-selection functions (e.g. those used in other TM papers [7, 17]), and found that they perform similarly and that all the conclusions obtained apply to them as well.

Metrics: In this section, we focus on the impact that each signature design has on performance. All the performance figures presented are normalized to those of a system with "perfect" signatures, i.e. one that has no false positives or false negatives and has a single-cycle access time. This signature is not implementable in bounded hardware, but provides an upper bound on conflict detection capabilities. The simulated system's memory latency was randomly perturbed, and we did multiple runs of each benchmark and configuration to obtain stable averages [1].

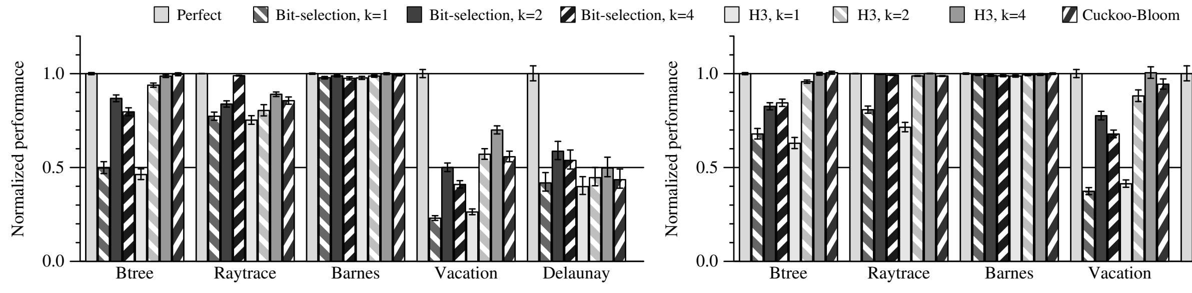
6.2. Results

For the sake of clarity and conciseness, we present a subset of the results we have obtained. The full set of results (with, for example, more signature sizes) can be found in a technical report [25]. Those results further corroborate the observations made in this paper.

True vs parallel Bloom signatures: Figure 7 shows normalized performance when using true and parallel Bloom signatures of four hash functions, with either bit-selection or H_3. In general, the performance differences between using a true and a parallel Bloom signature are quite small, and are higher for bit-selection than for H_3 (with a 8.2% versus a 3.6% mean difference), since H_3 hash functions create more uniform and uncorrelated distributions of the hash values. Recall that in our formal analysis we showed that parallel Bloom matches true Bloom when addresses are evenly distributed. Also, we can easily see how parallel Bloom signatures with H_3 perform better than their true Bloom counterparts whenever the differences are noticeable.

Implication 1: Since parallel Bloom signatures perform either equivalently or slightly better than true Bloom signatures and are more area-efficient, we recommend them and will focus on parallel Bloom signatures from now on.

Number and type of hash functions: Figure 8 clearly shows that the effect of increasing the number of hash functions depends strongly of the type of hash functions used.



(a) 512 bits

(b) 2048 bits

Figure 8: Performance of parallel Bloom signatures with different number and type of hash functions

For bit-selection, increasing the hash functions beyond 2 often degrades performance. This result is consistent with the findings in Bulk [7], in which bit-selection is used, and it is concluded that using more than two hash functions yields worse performance. However, we can see that when using H_3 hash functions, performance increases steadily when increasing the number of hash functions up to four. With eight H_3 hash functions (not shown for clarity) we observed performance drops marginally (1% less on average). Again, this is an effect of the higher quality of H_3 hashing.

In absolute terms, we can see how we can achieve a significantly higher performance for a given size if we use H_3 hash functions. For example, the best 2048-bit design with H_3 hash functions outperforms the best signature using bit-selection by 30% in Vacation and 27% in Delaunay.

Implication 2a: When using low-quality hash functions like bit-selection, we confirm that more than two hash functions does not help.

Implication 2b: We advocate using high-quality hash functions, like those from the H_3 family, that can generate many uncorrelated and equally distributed hash values, and using four or more such hash functions.

Impact of the number of cores: We now study a machine configuration with a small signature size of 256-bits and two H_3 hash functions, and a varying number of cores (keeping the other parameters of the system as shown in Figure 6b), thus making signatures critical to performance. Figure 9 shows the relative performances for CMPs with 8, 16, and 32 cores, both when broadcast is used, and when we use the directory protocol proposed in LogTM-SE instead. In general, we can see how insufficiently accurate signatures hurt performance much more as we increase the number of cores. For example, when the broadcast protocol is used, there is a a 27% slowdown in Vacation with 8 cores, and a 143% slowdown with 32 cores. Also, as we can see, the directory protocol is often effective in reducing the performance degradation caused by false positives (as fewer tests are performed), but signature size still needs to increase with the number of cores for the more demanding benchmarks. For example, the slowdowns in Vacation are 11% with 8 cores, and 51% with 32 cores.

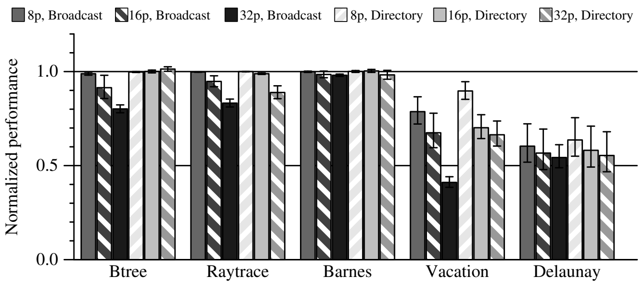


Figure 9: Performance with a 256-bit, two H_3 hash function signature with a varying number of cores

Implication 3: As the number of cores increases, signatures must be carefully designed to mitigate the increased potential for false positives, especially if broadcast coherence is used. However, using a directory (at L2 cache banks) enables larger TM systems without undue false positives, mitigating the need to increase signature hardware.

Performance of Cuckoo-Bloom signatures: To see how Cuckoo-Bloom signatures perform in a practical setting, we simulate them in the same conditions as Bloom signatures. We used signatures with the parameters shown in Table 1, keeping the 16-bit entries, with either 16 or 64 sets, to get sizes of 512 and 2048 bits. Figure 8 shows the performance impact of Cuckoo-Bloom signatures for the different benchmarks. Its performance is at least between the two and four hash function Bloom signatures, and for most benchmarks it outperforms the best bit-selection Bloom signature and is similar in performance to the four hash function H_3 Bloom signature. Furthermore, these results improve comparatively with bigger sizes (as we can hold an accurate representation of more elements).

Implication 4: Cuckoo-Bloom signatures can outperform parallel Bloom signatures, especially when Bloom signatures use one or two hash functions.

7. Related work

We review related work in Bloom filters and their application as signatures for transactional memory. Bloom filters were first proposed in 1970 [2]. Many variations on Bloom filters have been proposed. Counting Bloom filters allow deletions and can represent multisets [10]. Hash-table-based alternatives similar to Bloom filters have been proven to be more space-efficient than Bloom filters themselves [6, 19]. Bloom filters have been used in computer architecture for purposes other than conflict detection (e.g., for load-store queues [26] and early miss detection in L2 caches [11]).

Hardware implementations of Bloom filters are common in network applications. Broder and Mitzenmacher survey the theory and applications of Bloom filters in networks [4]. Dharmapurikar et al. describe a packet inspection system with hardware-implemented Bloom filters [9]. Often, these designs require efficient implementations of counting Bloom filters [3, 24].

The choice of hash functions in hardware Bloom filters is crucial, because complex functions may achieve better performance, but take more area. Ramakrishna et al. compare bit-selection, simple XORing and H_3 for address hash tables with real-life data, and conclude that only H_3 functions can achieve analytical performance in practice [23]. Earlier work by Ramakrishna shows how to achieve analytical performance with Bloom filters in a practical setting by using a universal hash function [22]. The H_3 family of hash functions was first described by Carter and Wegman [5]. Vandierendonck and De Bosschere describe two approaches to measure and improve the quality of specific XOR-based hash functions [28].

Several transactional memory systems have adopted signatures. VTM [21] uses a global signature (XF), implemented as a counting Bloom filter, to filter conflict tests after cache victimization and other rare events. Some software TM systems use signature-like structure with a single hash function, which can suffer many false positives [33]. Bulk [7] pioneered performing all conflict detection with local signatures. It implemented (what we refer to as) parallel Bloom signatures with bit-selection hashing. BulkSC [8] uses Bulk's structures to enforce sequential consistency. LogTM-SE [31] uses a similar signature implementation, but operates with a directory filter and adds summary signatures to support context switching and paging. Finally, SigTM [17] uses signatures in a manner similar to LogTM-SE, but uses the more expensive true Bloom signature implementation. None of these TM papers, however, provides an analysis of the performance and area requirements across signature design alternatives, as done in this paper.

8. Conclusions

Signature-based conflict detection is a promising approach in TM systems, as it enables transactions unbounded in size in a hardware-efficient manner, at the expense of a typically small performance hit. Multiple signature-based designs have been proposed so far, but the novelty and complexity of these systems left little room to cover in depth the different approaches to signature implementation.

This paper is the first to perform a detailed comparison of the performance and area of three signature design alternatives: *true Bloom, parallel Bloom,* and the newly-proposed *Cuckoo-Bloom*. We find, for example, that parallel Bloom signatures are preferred to true Bloom signatures, hashing via bit-selection is usually not sufficient, directory filtering can be valuable, and Cuckoo-Bloom signatures can outperform Bloom signatures.

Although the benchmarks used in the evaluation provide insight into signature behavior, future studies could use a richer set of benchmarks for a more thorough evaluation. Also, there is room for future work in designing hash functions. For example, they could randomly change when a conflict is detected to avoid repeated false conflicts, or even dynamically adapt to the workload.

Acknowledgments

We thank Dan Gibson, Marc de Kruijf, Milo Martin, David Wood and the Wisconsin LogTM and Multifacet groups for their comments. We thank the Wisconsin Condor project, the UW CSL, and Virtutech for their assistance. Sanchez is supported by a Vodafone Spain Scholarship and a Caja Madrid Fellowship. This work is supported in part by the National Science Foundation (NSF), with grants EIA/CNS-0205286, CCR-0324878, CNS-0551401, and CNS-0720565, as well as donations from Intel and Sun Microsystems. Hill has significant financial interest in Sun Microsystems. The views expressed herein are not necessarily those of our sponsors.

References

[1] A. R. Alameldeen and D. A. Wood. Variability in architectural simulations of multi-threaded workloads. In *Proc. of the 9th International Symposium on High-Performance Computer Architecture*, Feb. 2003.

[2] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, July 1970.

[3] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese. An improved construction for counting Bloom filters. In *Proc. of the European Symposium on Algorithms '06*, pages 684–695, 2006.

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

[5] J. L. Carter and M. N. Wegman. Universal classes of hash functions (Extended Abstract). In *Proc. of the 9th Annual ACM Symposium on Theory of Computing*, pages 106–112, 1977.

[6] L. Carter, R. Floyd, J. Gill, G. Markowsky, and M. Wegman. Exact and approximate membership testers. In *Proceedings of the 10th Annual ACM Symposium on Theory of Computing*, pages 59–65, 1978.

[7] L. Ceze, J. Tuck, C. Cascaval, and J. Torrellas. Bulk disambiguation of speculative threads in multiprocessors. In *Proc. of the 33rd Annual International Symposium on Computer Architecture*, June 2006.

[8] L. Ceze, J. Tuck, P. Montesinos, and J. Torrellas. BulkSC: Bulk enforcement of sequential consistency. In *Proc. of the 34th Annual International Symposium on Computer Architecture*, June 2007.

[9] S. Dharmapurikar, P. Krishnamurthy, T. Sproull, and J. Lockwood. Deep packet inspection using parallel Bloom filters. In *Proc. of the Symposium on High Performance Interconnects '03*, pages 44–51, Aug 2003.

[10] L. Fan, P. Cao, J. Almeida, and A. Z. Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Trans. Netw.*, 8(3):281–293, 2000.

[11] M. Ghosh, E. Őzer, S. Biles, and H.-H. S. Lee. Efficient System-on-Chip energy management with a segmented Bloom filter. In *Proc. of the 19th International Conference on Architecture of Computing Systems*, pages 283–297, March 2006.

[12] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. In *Proc. of the 31st Annual International Symposium on Computer Architecture*, June 2004.

[13] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the Twentieth Annual International Symposium on Computer Architecture*, pages 289–300, May 1993.

[14] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-way multithreaded SPARC processor. *IEEE Micro*, 25(2):21–29, 2005.

[15] J. R. Larus and R. Rajwar. *Transactional Memory*. Morgan & Claypool Publishers, 2006.

[16] M. M. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *Computer Architecture News*, pages 92–99, Sept. 2005.

[17] C. C. Minh, M. Trautmann, J. Chung, A. McDonald, N. Bronson, J. Casper, C. Kozyrakis, and K. Olukotun. An effective hybrid transactional memory system with strong isolation guarantees. In *Proc. of the 34th Annual International Symposium on Computer Architecture*, June 2007.

[18] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. LogTM: Log-based transactional memory. In *Proc. of the 12th International Symposium on High-Performance Computer Architecture*, pages 254–265, Feb 2006.

[19] A. Pagh, R. Pagh, and S. S. Rao. An optimal Bloom filter replacement. In *Proc. of the 16th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 823–829, 2005.

[20] R. Pagh and F. F. Rodler. Cuckoo hashing. In *Proc. of the 9th Annual European Symposium on Algorithms*, pages 121–133, 2001.

[21] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing transactional memory. In *Proc. of the 32nd Annual International Symposium on Computer Architecture*, pages 494–505, 2005.

[22] M. V. Ramakrishna. Practical performance of Bloom filters and parallel free-text searching. *Commun. ACM*, 32(10):1237–1239, 1989.

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

[24] E. Safi, A. Moshovos, and A. Veneris. L-CBF: a low-power, fast counting Bloom filter architecture. In *Proc. of the 2006 International Symposium on Low Power Electronics and Design*, pages 250–255, 2006.

[25] D. Sanchez. Design and Implementation of Signatures for Transactional Memory Systems. Technical Report CS-TR-2007-1611, University of Wisconsin-Madison, Aug. 2007.

[26] S. Sethumadhavan, R. Desikan, D. Burger, C. R. Moore, and S. W. Keckler. Scalable hardware memory disambiguation for high ILP processors. In *Proc. of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, 2003.

[27] D. Tarjan, S. Thoziyoor, and N. P. Jouppi. CACTI 4.0. Technical Report HPL-2006-86, HP Labs, 2006.

[28] H. Vandierendonck and K. D. Bosschere. XOR-based hash functions. *IEEE Trans. Comput.*, 54(7):800–812, 2005.

[29] J.-M. Wang, S.-C. Fang, and W.-S. Feng. New efficient designs for XOR and XNOR functions on the transistor level. *IEEE Journal of Solid-State Circuits*, 29(7):780–786, July 1994.

[30] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proc. of the 22nd Annual International Symposium on Computer Architecture*, pages 24–36, June 1995.

[31] L. Yen, J. Bobba, M. R. Marty, K. E. Moore, H. Volos, M. D. Hill, M. M. Swift, and D. A. Wood. LogTM-SE: Decoupling hardware transactional memory from caches. In *Proc. of the 13th International Symposium on High-Performance Computer Architecture*, pages 261–272, Feb 2007.

[32] A. Zeichick. One, two, three, four: A sneak peek inside AMD's forthcoming quad-core processors. Technical report, AMD, Jan. 2007.

[33] C. Zilles and R. Rajwar. Transactional memory and the birthday paradox. In *Proc. of the 19th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 303–304, 2007.