Notary: Hardware Techniques to Enhance Signatures

Luke Yen*, Stark C. Draper+, and Mark D. Hill*+
*Department of Computer Sciences, +Department of Electrical and Computer Engineering
University of Wisconsin—Madison, WI U.S.A.
lyen@cs.wisc.edu, sdraper@ece.wisc.edu, markhill@cs.wisc.edu

Abstract

Hardware signatures have been recently proposed as an efficient mechanism to detect conflicts amongst concurrently running transactions in transactional memory systems (e.g., Bulk, LogTM-SE, and SigTM). Signatures use fixed hardware to represent an unbounded number of addresses, but may lead to false conflicts (detecting a conflict when none exists). Previous work recommends that signatures be implemented with parallel Bloom filters with two or four hash functions (e.g., H_3).

Two problems exist with current signature designs. First, H_3 implementations use many XOR gates. This increases hardware area and power overheads. Second, signature false positives can result from conflicts with signature bits set by private memory addresses that do not require isolation.

This paper develops Notary, a coupling of two signature enhancements to ameliorate these problems. First, we use address entropy analysis to develop Page-Block-XOR (PBX) hashing and show it performs similar to H_3 at lower hardware cost. Second, we introduce a privatization interface that explicitly allows the programmer to declare shared and private heap memory allocation. Privatization reduces false conflicts arising from private memory accesses and can lead to a reduction in the signature size used.

Results from custom transistor-level layouts of H_3 and PBX, along with full-system simulation of a 16-core chip-multiprocessor implementing LogTM-SE, show (a) PBX hashing performs similar to H_3 hashing while requiring up to 24% less area and 4.7% less power overhead and (b) privatization can improve execution time by up to 86% (by reducing false conflicts by up to 96%).

1. Introduction

The reality of multi-core chip-multiprocessors (CMPs) is here. Major vendors (e.g., Intel, AMD, IBM, and Sun) have released products incorporating two to eight cores on a single chip [3,20,22]. However, most software has not matured sufficiently to take advantage of the increased hardware resources. A primary reason is that parallel programming is a difficult task. *Transactional Memory* (TM) [14,21] is emerging as a programming paradigm that eases the task of writing parallel programs. TM introduces atomic blocks to replace uses of locks in multi-threaded parallel programs. The promise is deadlock-free, composable code. In essence, programmers declare which regions of code they want to execute atomically, and the underlying TM system enforces these requests.

High-performance TM systems execute multiple transactions concurrently and simultaneously commit only those that do not conflict. A *conflict* occurs when two or more concurrent transactions perform an access to the same memory address and at least one of the accesses is a write. In order to detect conflicts, the TM system must record the set of addresses that a transaction reads (the *read-set*) and writes (the *write-set*). Recently, *signatures* have been proposed as a hardware mechanism to track read- and write-sets in a compact manner. Signatures were first proposed for Bulk [9]. Several recent systems have also adopted its use, including BulkSC [10], LogTM-SE [45], and SigTM [28]. These systems implement signatures with per-thread hardware Bloom filters [7]. Read- and write-sets are tracked by inserting the addresses of transactional loads and stores into the read and write signatures, respectively. Address lookups are implemented either as a test operation (LogTM-SE, SigTM) or by intersecting two signatures (Bulk, BulkSC). A test or intersection operation may signal a conflict when none existed (a *false positive*), but never misses a conflict (no *false negatives*). Finally, signatures are cleared on commit and abort.

False conflicts arising from false positives determine the effectiveness of signatures as a mechanism for conflict detection. As Section 6 shows, small signatures can increase execution time by up to seven times. Increasing signature size usually reduces false conflict rates (much as increasing cache size reduces miss rates), but at some point this solution becomes prohibitively expensive. For example, when using 64kb signatures, there is a 1330% increase in area and a 106% increase in power over 1kb signatures (details in Section 3.6). Moreover, larger signatures increase the overheads when signatures move (e.g., Bulk's signature broadcasts [9] and LogTM-SE's saving/restoring signatures [45]). Finally, depending on the implementation, smaller signatures may result in lower access latencies than larger signatures. For these reasons, *this paper will focus on effectively using signature hardware once its size is fixed*.

First, we review signature background in Section 2. In particular, Sanchez et al. [33] advocate that signature implementations should use parallel Bloom filters (not true Bloom filters) with two or four hash functions (e.g., from the H_3 family [8,31]).

This work is supported in part by the U.S. National Science Foundation 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.

Second, in Section 3 we develop the ***Page-Block-XOR (PBX)*** hash function that we will show performs similar to the H_3 hash functions at much less hardware cost. Specifically, we examine address bit entropy [36]. We find that using a single-level of XOR gates operating on non-overlapping fields of an address' page and cache-index bit-fields works well, and name this PBX. In particular, we find that PBX improves on H_3 hashing by using up to 24% less area and 4.7% less power.

Third, in Section 4 we develop a ***privatization interface*** that reduces false conflicts by giving programmers the ability to inform the TM system of addresses that *cannot* cause transaction conflicts. The TM system uses this information to keep these addresses out of signatures. Thereby fewer signature bits are set that might result in false conflicts. We show that privatizing the stack has a negligible effect, so we instead concentrate on heap data by supporting private and shared memory allocators that seek to put private and shared data on different pages.

Finally, in Section 5 we present our evaluation methodology, and in Section 6 we evaluate the combination of our ideas that we call ***Notary***. Results for a 16-core CMP implementing LogTM-SE [45] show (a) PBX hashing performs similar to H_3 hashing (and requires less hardware) and (b) privatization can improve execution time by up to 86% (by reducing false conflicts by up to 96%). In Section 6.3 we discuss how Notary can be implemented in any signature-based TM system and in signature-based non-TM systems.

2. Signature Background

2.1 Prior signature systems

Signatures are a hardware mechanism to concisely represent a set of addresses without the possibility of false negatives. They are usually implemented as Bloom filters. Signatures for TM were first proposed by Ceze et al. [9] for use in Bulk, which broadcasts signatures in order to support both a hardware TM (HTM) system and a speculative multi-threading system. Other HTM systems have also used signatures to detect conflicts amongst concurrent transactions.

Signatures are also used for purposes besides TM and speculative multi-threading. Examples include BulkSC [10], which uses hardware-defined (implicit) transactions in order to track violations of the sequential consistency memory model. The SoftSig framework [39] defines a general software programming interface for signatures. Atom-Aid [23] uses signatures to detect potential atomicity violations amongst implicit transactions. Finally, signatures can be used in the design of race-recording mechanisms for multi-threaded programs, and examples include DeLorean [29] and Rerun [15].

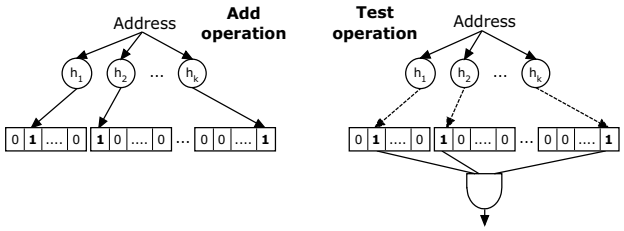

Figure 1: Parallel Bloom signature design

2.2 Prior signature results

Although numerous papers have proposed systems that use signatures, Sanchez et al. [33] focus exclusively on important implementation and performance issues for TM systems. They present four conclusions stemming from their analysis of signature design and performance: (1) signature false positives are detrimental to performance (e.g., up to 50% performance degradation with 512b signatures), (2) parallel Bloom signatures should be used for implementation efficiency, (3) H_3 hashing should be used in favor of the lower-quality bit-selection hash function, and (4) signatures should use two or more hash functions. Our paper's focus is on reducing signature implementation overheads and on a privatization interface for signatures.

Parallel Bloom signatures partition a single, logical signature, and each partition is accessed independently by a hash function. Figure 1 illustrates this signature and its two operations (add and test) for k hash functions. During an add operation, each of the k hash functions hashes the input address and sets one bit in its designated signature partition. A test operation indicates the presence of an address only if all individual hash functions indicate its partition's bit is set during lookup.

2.3 H_3 hash functions

The H_3 class of hash functions creates uniformly distributed hash values [8,31]. For a straightforward

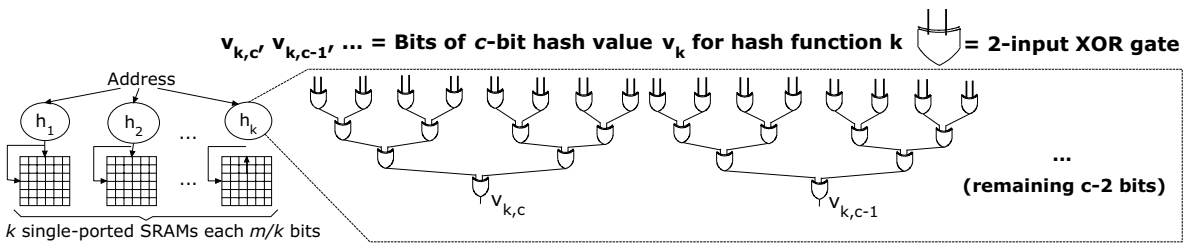

Figure 2: Bloom signature implementation with k H_3 hash functions, c-bit hash values, and a m-bit signature

implementation of H_3 using fixed Boolean matrices to select address bits, on average half of the input address bits can affect a given bit of a hash value. Figure 2 illustrates this for 32-bit addresses, in which sixteen address bits produce each bit of a *c*-bit hash value. While flexible, this implementation uses a large number of XOR gates, even after optimizations [40]. In general, given fixed Boolean matrices, the number of 2-input XOR gates required to implement *k* hash functions producing *c*-bit hash values is:

$$\text{Num XOR} = \sum_{j=1}^{k} \sum_{i=1}^{c} \left\lceil \frac{\text{Number of 1s in column } i \text{ of matrix } j}{2} \right\rceil$$

$$\approx \frac{\text{address length in bits}}{4} \times c \times k$$

For example, assume 2kb signatures with two hash functions (*k*=2) are used. Each hash generates a 10-bit hash value (*c*=10). If sixteen bits from a 32-bit address (e.g., sixteen ones in each column of the Boolean matrix) generate a bit of a hash value, then a total of 160 XOR gates per signature will be required. The overhead increases if processor cores support multiple thread contexts and each context uses separate signatures. Similar overheads can be calculated using XOR gates with higher fan-in (e.g., 4-input XORs). We present quantitative analysis of the additional area and power overheads of H_3 hash functions in Section 3.6.

2.4 XOR hashing

Our proposed PBX hashing, detailed in Section 3, is an XOR-based hashing function. This section describes prior uses of and alternatives to XOR hashing. In memory hierarchy designs, hash functions are primarily used to reduce cache, bank, or DRAM row-buffer conflicts. In particular, both XOR-based hash functions [12,35,46] and polynomial hash functions [32] have been proposed. Both are successful at reducing memory storage conflicts.

Kharbutli et al. [18,19] propose two alternative hash functions to XOR that reduce the probability of bad hash values: prime modulo hashing and odd-multiplier displacement hashing. While efficient implementations exist (e.g., using adders and selectors for prime modulo indexing and adders for odd-multiplier displacement), it can require modifications to existing hardware structures (e.g., additional bits in each hardware TLB entry) or additional hardware (adders). In contrast, PBX uses only a small number of XOR gates and no modifications to hardware (e.g., to L1 cache, TLB).

Vandierendonck et al. [40] provide a detailed examination of XOR hashing by using the concepts of null and column spaces from linear algebra. They show that elementary matrix operations (replacing and swapping columns) can minimize the fan-in and maximum fan-out of XOR gates, and also discuss polynomial hash functions (both reducible and irreducible [32]). Both H_3 and PBX benefit from their results, by reducing the fan-in and fan-out of the XOR gates used in implementations.

3. PBX: Using Entropy to Reduce H_3 Costs

3.1 Motivation

H_3 hashing requires many XOR gates, which increases the area, power, and latency of signature implementations. Since most programs do not use the entire virtual address space, some address bits exhibit more randomness than other bits. This suggests that it may be unnecessary for H_3 to use multi-level XOR trees to produce random hash values, as the input bits already have randomness. The insight behind PBX is that **if the input bits have sufficient randomness and those bits are used as inputs to the hash functions, then random hash values result.** PBX combines a simple yet powerful technique of selecting the input bits to the hash functions with a much simpler hash function. This reduces the overall area and power overheads compared with H_3. Moreover, simulations show that the randomness in our workloads' addresses is far from the maximum, and the most random bits are localized to less significant bit-fields in the address.

3.2 Entropy

We calculate the randomness of addresses using *entropy* [36], an important measure of randomness. Entropy is a measure of the uncertainty of a random variable *x*, and is calculated as:

$$Entropy = -\sum_{i=1}^{N} p(x_i)\log_2 p(x_i)$$

where $p(x_i)$ is the probability of the occurrence of value x_i, N is the number of sample values the random variable x can take on, and *Entropy* is the amount of information required on average to describe an outcome of the random variable *x*. The units are bits. For example, if every bit pattern in a *n*-bit field occurs equally often, entropy is *n* bits. At the other extreme, if only a single pattern is possible, the entropy is zero bits. All other distributions have entropy greater than zero and less than *n* bits.

To estimate the entropy of a workload's addresses, we use the following definitions in our calculations. Let $\chi = \{0, 1, 2, \ldots, 2^n - 1\}$, where n is the length of the address field. Our data consists of d addresses $\{a_j\}_{j=1}^{d}$, where $a_j \in \chi$ for all $j \in \{1, \ldots, d\}$. We use the empirical entropy

of the data sequence as our estimate of entropy, which we calculate as:

$$Entropy = -\sum_{i=1}^{2^n} \hat{p}(x_i)\log_2 \hat{p}(x_i)$$

$$\hat{p}(x_i) = \frac{1}{d}\sum_{j=1}^{d} I(a_j = x_i)$$

where $I(a_j = x_i) = 1$ if $(a_j = x_i)$, 0 otherwise and

$$0\log 0 = 0$$

We choose to calculate entropy over the data addresses of an entire workload's execution (with some filtering, described below). Although representative, this might hide entropy changes resulting from program phase changes. Nevertheless, we find our choice yields good overall entropy trends that PBX exploits. Our address filtering removes addresses that do not operate on signatures. For example, we ignore supervisor-level addresses, as these are never added to signatures and bypass signature tests. We calculate the entropy of load and store addresses separately, and differentiate between entropies of virtual and physical addresses. We show entropy results for physical memory accesses inside transactions. Similar trends exist for the virtual addresses.

Our entropy calculations assume 32-bit virtual and physical addresses, a cache-block size of 64B, and a page size of 4kB. We assume addresses are little-endian, such that the least significant bit is bit 0. In our entropy results, we represent the ending and starting bit indices of a bit-field of the memory address as [end:start]. The bit-field representing the entire input address is [31:0]. Using 64-bit virtual and physical addresses would not significantly affect our entropy results, since our workloads have working sets that do not exceed 32-bit addresses. We discuss entropy results of larger workloads in Section 3.5. Finally, increasing the interleaving of a program's address space in physical memory would also likely increase the entropy of the higher-order physical address bits.

We calculate two types of entropy for each workload, one on the entire input address (*global* entropy), and the other on bit-fields of the input address (*local* entropy). For the parameters given above this means the maximum global entropy value for any of our workloads is 26 bits (e.g., the address bits remaining after masking the block offset[1]). These two types of entropy give the upper bound of a workload's address entropy and local entropy trends within sub-fields of the input address (e.g., among the lower-order or the higher-order bits).

1. Specific to our HTM implementation. Other TMs may choose different address masking granularities.

3.2.1 Previous uses of entropy

Entropy has previously been used as a measure of randomness in computer architecture research. For example, Hammerstrom et al. [13] use entropy to measure the overheads of addressing memory in an instruction-set architecture. Park et al. [30] describe how address bits can be transferred with smaller bit overheads through the use of a Base Register Cache. This mechanism works because higher-order bits have lower entropy and many programs exhibit spatial locality. Becker et al. [6] extend this work by using Dynamic Huffman Coding to further improve bit compression. Citron et al. [11] describe a mechanism which compacts and expands both addresses and data values over a shared bus. Ballapuram et al. [5] propose low-power TLB designs through the calculation of the entropy of virtual page numbers of heap, global, and stack addresses. We differ from their work in the following ways. First, we examine entropy trends within an address. Our focus is on the average entropies across the entire workload, rather than on changes in entropy values over time. Second, we believe we are the first to incorporate the idea of entropy in the selection of the input bits for XOR hashing.

3.3 Results

In this section we present entropy results gathered from a suite of ten transactional workloads running on top of a simulated 16-core CMP, with details in Section 5.

Entropy results for our workloads are shown in Figure 3. We first describe the results for the load and store global entropies ("LD/ST Global" lines). First, we note that our workloads' global entropy values are much lower than the maximum value of 26 bits. For our workloads the global entropy values are between 1-16 bits. Second, with the exceptions of Raytrace and BIND the store global entropy value is larger than the load global entropy value. In Raytrace and BIND, the set of transactional store addresses that is frequently accessed is small.

Local entropy is calculated by using a bit-window that moves from lower- to higher-order address bits. This bit-window is 16 bits wide, since global entropy values never exceed 16 bits, and a bit-window of this size allows local entropy values to be close to or equal global entropy values. We vary the starting bit index of this window to capture local entropy changes as the window moves towards higher-order bits. This is done by varying N_{skip}, a variable representing the number of lower-order bits skipped before starting the bit-field. Using our notation, this bit-field is $[(N_{skip}+21):(N_{skip}+6)]$, where N_{skip} varies from 0-10.

The results for local load and store entropies are the lines labeled "LD/ST Local" in Figure 3. To help understand these results, we focus on BTree's results (the upper-left subplot). For load addresses, as the bit-window moves towards higher-order bits the local entropy decreases from an average of about 9 bits to 4 bits. Similarly, for store

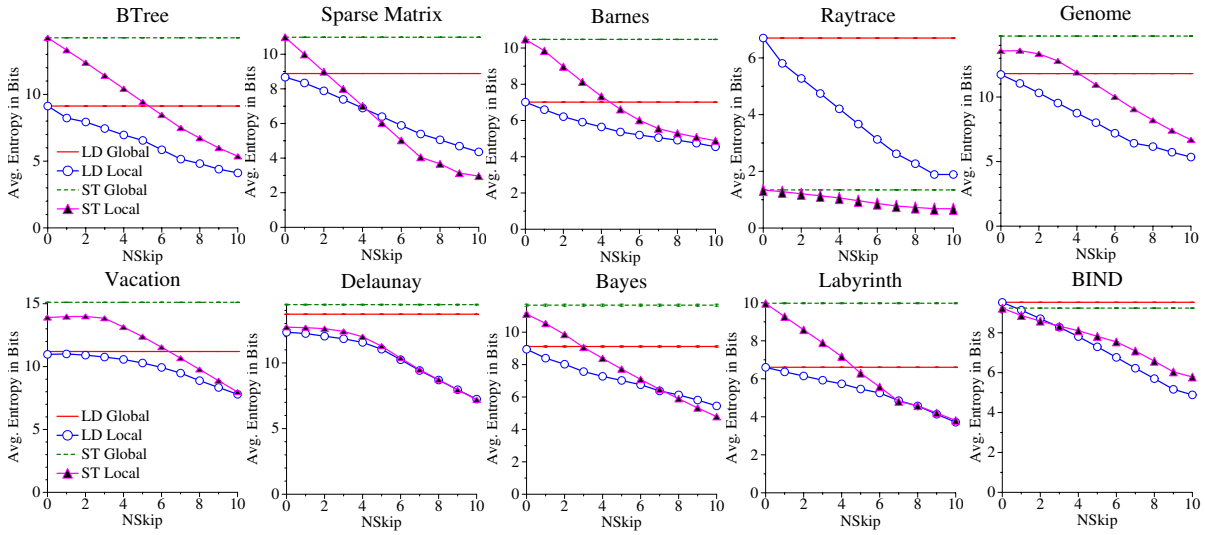

Figure 3: Entropy of transactional load and store physical addresses

addresses entropy decreases from about 14 bits to 5.5 bits. For the majority of the workloads the global entropy values can be captured by the least significant 16 bits of the block address. For the remaining workloads this is not true, and the local entropy lines do not intersect with the global entropy lines (in Genome, Vacation, Delaunay, and Bayes). This is because not all of the global entropy value can be captured by a 16-bit window. The remaining can be captured if the window is increased to 26 bits.

The entropy trends all result in smooth, monotonically decreasing curves. This occurs for two reasons. First, our transactional workloads have small working set sizes, which limits the number of pages accessed and reduces local entropy for higher-order bits. Second, our entropy results are gathered on a newly booted simulated machine with little virtual memory activity. In systems with more processes and virtual memory activity the higher-order physical address bits are likely to exhibit more entropy.

In summary, the key finding of these results is that for our workloads, local entropies monotonically decrease as we examine higher-order bits.

3.4 Exploiting entropy

We propose PBX (Page-Block-XOR) hashing to exploit entropy characteristics. PBX is motivated by three findings: (1) lower-order bits have the most entropy, and therefore are the best candidates for hashing, (2) XORing two bit-fields produces random hash values, and (3) bit-field overlaps lead to higher false positives due to correlation between the bit-fields. Our discussion will reference Figure 3 and Figure 4, and PBX signatures using two hash functions. The pairs of numbers in parentheses in Figure 4 indicate the bits XORed together to produce each hash bit.

We now discuss these three findings. First, our local entropy results indicate lower-order bits are more random (higher entropy) than higher-order bits. This is intuitive because programs tend to exhibit spatial locality. According to Figure 3, the highest local entropy comes from the bit-window with zero N_{skip} bits. We found through further analysis that for our workloads address bits [26:6] are the most random (and bits [31:27] are mainly constant).

Second, previous research has shown that XOR hashing of bit-fields produce random hash values [12,32,35,46]. We, like others, find that XORing two bit-fields produces random hash values. To achieve this we partition the bit-field [26:6] into two bit-fields, one of length 11 bits (bit-field [16:6]), and the other of length 10 bits (bit-field [26:17]). Other partitions are possible, but may not produce all possible hash values (e.g., at the extreme the bit-field [6:6] XORed with another bit-field produces at most two different hash values). Since bits [16:6] are traditionally known as the cache-index bits in caching literature we call this bit-field the cache-index bits. However the size of this bit-field is not tied to any specific cache configuration in our CMP. Similarly, we call bit-field [26:17] the physical page number (PPN) since these bits are traditionally part of the PPN derived during address translation. However like the cache-index bit-field this is not tied to any specific parameter (e.g., the page size) in our CMP. It is important to use most or all of the bits in the bit-fields as inputs to our PBX hashes, as more bits lead to higher probability that all

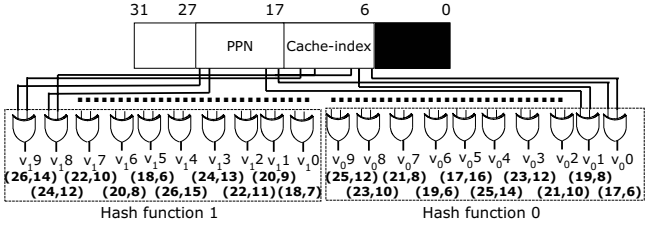

Figure 4: Bit-fields used for PBX

possible hash values will be produced. We verified that our PBX implementation produces random hash values by analyzing the PBX hash matrices used to select the hash bits (implied by the connections in Figure 4). We confirmed they can produce all possible hash values.

Third, our cache-index and PPN bit-fields do not overlap. This is beneficial because overlaps can cause correlation in the resultant hash values and reduce the range of hash values produced. Other simulation results (not shown) confirm that correlation can be detrimental to the performance of PBX.

In summary, as shown in Figure 4, our PBX signature configurations use two non-overlapping, consecutive bit-fields corresponding to the cache-index and PPN bit-fields. Since signatures use two hash functions, we distribute even-numbered bits in the bit-fields to one hash function, and the odd-numbered bits to the other hash function.

3.5 Generalizing to other workloads

The methodology presented in this section to obtain random address bit-fields applies to other workloads (with bigger working sets) and to larger addresses (e.g., 64-bits). We performed the same analysis as in Figure 3 using Wisconsin's Commercial Workload suite (Apache, Zeus, OLTP, and JBB) [2]. We found load global entropy values range from 13-15 bits and store global entropy values range from 13-17 bits. At N_{skip}=10, load local entropy values range from 9-11 bits, and store local entropy values range from 7-13 bits. The higher-order bits exhibit more randomness than in our transactional workloads. Thus the sizes of the cache-index and PPN bit-fields may be increased to capture these bits.

Finally, PBX's bit-field selection can be part of a feedback mechanism for its implementation. Initial runs of a newly-developed TM program can use pre-defined default bit-fields for the PBX signature. As entropy information is gathered during subsequent executions, the programmer or runtime system can more carefully tune the selection of bit-fields to enable better signature performance.

3.6 Savings in area and power overhead

PBX translates to lower area and power overheads than H_3. To quantify this, we implement custom transistor-level layouts of different PBX and H_3 hash functions using Cadence's Virtuoso design tool (version 5.10.41). Our hash designs use the optimized transistor-level XOR proposed by Wang et al. [41]. The designs incorporate PMOS transistors having a width of 1200nm and length of 400nm and NMOS transistors having width 600nm and length 400nm. We use linear scaling to scale the area estimates from 400nm to 65nm technology. Because many variables are involved in scaling power with technology, we do not scale any of our power numbers from our base 400nm technology. We expect the relative power results to hold if the other variables affecting power also scale with smaller technology nodes. The area and power of the Bloom filters are estimated using CACTI 4.2 [38], and we use 65nm technology for area results. The Bloom filters are implemented as SRAM banks with one read and one write port, and a word size of eight bytes. Due to limitations of CACTI, we only collect results for signature configurations 1kb and larger.

Table 1: Area overheads (in mm^2) of H_3 and PBX sig.

Sig. config.	Bloom filter	H_3 hash	PBX hash	H_3 sig.	PBX sig.	% savings for PBX sig.
2kb, k=2	1.80e-2	4.70e-3	2.80e-4	2.30e-2	1.80e-2	22
2kb, k=4	2.70e-2	8.10e-3	4.70e-4	3.50e-2	2.70e-2	23

Table 2: Power overheads (in W) of H_3 and PBX sig.

Sig. config.	Bloom filter	H_3 hash	PBX hash	H_3 sig.	PBX sig.	% savings for PBX sig.
2kb, k=2	1.10e-1	5.80e-3	5.68e-4	1.16e-1	1.11e-1	4.3
2kb, k=4	1.80e-1	1.04e-2	1.02e-3	1.90e-1	1.81e-1	4.7

3.6.1 Overheads of PBX compared to H_3

The area and power results of PBX and H_3 for 2kb signatures with two and four hash functions are presented in Table 1 and Table 2, respectively. We defer comparing the signature overheads with existing core designs until Section 3.6.2. The overheads are broken down according to the signature's components: the Bloom filter itself (column 2), and those from the hash function (columns 3 and 4). The overall overheads for H_3 and PBX signatures are summarized in columns 5 and 6, respectively. Our discussion focuses on signatures with two hash functions. Overall, PBX hash functions are 22% smaller than the same size signature implemented with two H_3 hash functions. H_3 hash trees consume approximately 20% of the total H_3 signature area overhead, while PBX hash functions only consume 1.6% of the total PBX signature area overhead.

The power results of 2kb H_3 and PBX signatures are shown in Table 2. Overall, 2kb PBX signatures (with two hash functions) use 4.3% less power than H_3 signatures. H_3 hash trees account for 5% of the H_3 signature's total power. However, PBX hash trees consume less than 1% of the total PBX signature's power.

In summary, signatures which use PBX instead of H_3 hash functions require less area and save power (up to 23% savings for area and 4.7% savings for power). The savings increase as more hash functions are used.

Table 3: Area and power overheads of PBX signatures

	Area (mm^2)			POWER6, 47mm^2 core		Niagara, 13mm^2 core		Power (W)		
Sig. (k=2)	Bloom filter	PBX hash	PBX sig.	PBX area (mm^2)	% core area for sig.	PBX area (mm^2)	% core area for sig.	Bloom filter	PBX hash	PBX sig.
1kb	1.40e-2	2.40e-4	1.40e-2	5.60e-2	0.12	2.20e-1	1.65	9.20e-2	5.12e-4	9.25e-2
2kb	1.80e-2	2.80e-4	1.80e-2	7.20e-2	0.15	2.80e-1	2.13	1.10e-1	5.68e-4	1.11e-1
4kb	3.20e-2	3.10e-4	3.20e-2	1.30e-1	0.27	4.90e-1	3.78	1.10e-1	6.27e-4	1.11e-1
8kb	6.20e-2	3.10e-4	6.20e-2	2.50e-1	0.53	9.50e-1	7.33	1.40e-1	6.83e-4	1.41e-1
16kb	7.60e-2	3.50e-4	7.60e-2	3.00e-1	0.65	1.17e0	8.98	1.60e-1	7.39e-4	1.61e-1
32kb	1.00e-1	3.60e-4	1.00e-1	4.00e-1	0.85	1.54e0	11.82	2.00e-1	7.95e-4	2.01e-1
64kb	2.00e-1	4.10e-4	2.00e-1	8.00e-1	1.70	3.07e0	23.63	1.90e-1	8.55e-4	1.91e-1

3.6.2 Overheads of larger PBX signatures

Although larger signature sizes reduce false positives, they do so at the cost of increased area and power. We calculate the area and power overhead trends of different PBX signatures ranging from 1kb to 64kb, all using two hash functions. The results are shown in Table 3. First, we examine the area and power overheads of different signature configurations relative to each other (columns 4 and 11, respectively). For example, increasing the signature size from 1kb to 4kb increases the signature's area overhead by 129% and power by 20%. Using 16kb signatures instead of 4kb signatures increases the signature's area overhead by 138% and power by 45%. Finally, increasing from 16kb to 64kb signatures increase area by 163% and power by 19%. Overall, larger signatures increase both area and power overheads.

Second, we compare signature area overheads relative to the area of two current micro-processor designs, the IBM POWER6 [22] and the Sun Niagara [20]. POWER6 is implemented in 65nm technology. The chip itself is 341mm^2, is dual-core, and each core is 47 mm^2 (measured from a die photo) and supports two SMT thread contexts. Niagara is implemented in 90nm technology. The chip area is 379mm^2, is eight-core, and each core is 13mm^2 (measured from a die photo) and supports four thread contexts. We calculate the total per-core area overhead consumed by the PBX signatures after scaling for technology (65nm or 90nm). We assume each thread context has a set of read and write signatures. We then compute the percentage of per-core area consumed by the per-core PBX signatures. The results are shown in columns 5-8 of Table 3. From the table it is evident that for the larger POWER6 cores PBX signatures do not consume a significant fraction of the core's total area. The largest 64kb signatures consume only 1.7% of the core's total area. However, this trend does not hold for the smaller Niagara core. Although 1kb signatures consume only 1.65% of core area, this overhead increases to 24% with 64kb signatures. Thus, with smaller core designs it is important to carefully consider the area overheads of using larger signature sizes.

Although PBX reduces signature implementation overheads, it does not mitigate false conflicts arising from signature bits set by private memory references. We now examine how Notary's technique of privatization reduces this type of false conflict.

4. Notary's Technique of Privatization

4.1 Motivation

Previous signature proposals do not associate high-level semantic information with inserted addresses. All addresses are treated with equal importance. However, data for many multi-threaded programs can be segregated into private and shared, according to whether the data is accessible to a single thread or shared amongst more than one thread. The process of performing this categorization is called privatization [37]. With information about private and shared memory accesses, the TM system has the opportunity to only isolate shared addresses and drop isolation on private addresses. Fewer addresses are inserted in signatures, and fewer signature bits will be set. Thus the probability of false positives on lookups will decrease. Although conceptually simple, prior research has not described how to implement this support for HTMs to handle memory objects of arbitrary sizes.

4.2 Methods and results

In order to illustrate the performance benefits of privatization, we execute a micro-benchmark containing a constant number of fixed-sized transactions, and only vary the fraction of the transaction's memory requests that are

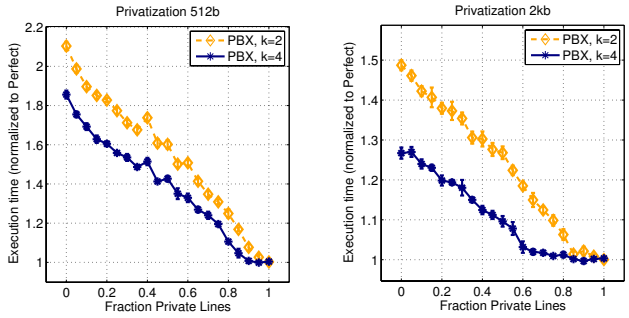

Figure 5: Privatization micro-benchmark

Table 4: Notary's privatization programming interface

Privatization function	Usage
`shared_malloc(size)`, `private_malloc(size)`	Dynamic allocation of shared and private memory objects.
`shared_free(ptr)`, `private_free(ptr)`	Frees up memory allocated by shared or private allocators.
`privatize_barrier(num_threads ,ptr, size)`, `publicize_barrier(num_threads , ptr, size)`	Program threads come to a common point to privatize or publicize an object. Must be used outside of transactions.

privatized (from 0.0 to 1.0, with 1.0 denoting all requests being privatized). We examine two different signature sizes (512b and 2kb) and two different numbers of hash functions (two and four). The micro-benchmark results are shown in Figure 5.

Focusing on the results for the 2kb signatures, we make several observations. First, privatization can greatly improve execution time for signatures. For example, with two hashes, 50% privatization leads to a 15% improvement in normalized execution time (compared to no privatization), and increases to a 32% improvement for 95% privatization. Second, the improvement benefits are still significant even after using more hash functions. For four hash functions, there is a 13% improvement in normalized execution time for 50% privatization, and 20% improvement for 95% privatization. Similar trends exist for the smaller 512b signature. We defer examining the results of our macro-benchmarks until Section 6.2.

4.3 Exploiting privatization

4.3.1 Prior TM privatization approaches

Researchers have proposed programmer-visible privatization language constructs, but these constructs have been targeted for software TM systems (STMs) or hybrid TMs. Little research has been done to show how HTMs can take advantage of privatization for data structures of arbitrary sizes.

For example, Scott et al. [34] describe the use of four varieties of pointers which guard accesses to transactional objects. Two of these pointer types, `sh_ptr<T>` and `un_ptr<T>`, denote transactionally shared and transactionally private data, respectively. Furthermore, researchers recently proposed a transactional version of OpenMP [27], which support the additional keywords `exclude` and `only` to denote which variables to exclude and track for conflict detection. Similarly, OpenTM [4] supports the keywords `private` and `shared`.

A possible implementation of these interfaces for HTMs is for the compiler to produce code using special load and store instructions for private object accesses. Then the HTM system can elide transactional isolation for these instructions. However special instructions can increase the complexity of memory instructions in already-complex instruction set architectures (e.g., loads and stores to special ASIs in SPARC). Furthermore, compilers may not guarantee which, if any, memory accesses can be converted to privatized memory accesses, especially with the use of pointers in a program.

Abadi et al. [1] expose the functions `protect()` and `unprotect()` to the programmer to denote when C# objects are safe to use inside and outside of transactions, respectively. This interface is functionally similar to Notary's barriers (described below). Matveev et al. [26] propose adding a Virtual Memory Filter to cores and having the programmer declare a range of memory locations to be transactional. This interface, although more general, is functionally similar to Notary's privatization interface. Both interfaces were concurrently proposed with Notary.

4.3.2 Notary's privatization support

Notary's privatization is implemented through a combination of compiler, library/runtime, operating system (OS), and minimal hardware support. The programmer invokes privatization through a set of library functions, and defines which objects are private or shared. Both static and dynamic variants of privatization are supported. Objects do not change sharing status in static privatization, whereas changes can occur in dynamic privatization.

The process of marking objects as private must be done with care, as mistakenly marking objects as private can lead to program errors. If there is any doubt on an object's sharing status, the programmer should mark it as shared. This might lead to performance degradations but not to correctness problems. Table 4 summarizes Notary's

```
s_array = (int*)shared_malloc(arraysize*sizeof(int));
xact_begin;
p_array = (int*)private_malloc(arraysize*sizeof(int));
// no conflict detection on p_array
p_array[i] = myid;
// detect conflicts on s_array
s_array[i] = myid*10;
// delete p_array
private_free(p_array);
...
xact_end;
// This might violate strong atomicity
s_array[j]=myid*15;
// delete s_array
shared_free(s_array);
```

(a) private and shared memory allocation

```
x = (int*) private_malloc(arraysize*sizeof(int));
// use x as private variable
...
// transition x from private to shared
publicize_barrier(numthreads,x,arraysize*sizeof(int));
xact_begin;
// use x as shared variable
x[i] = myid;
...
xact_end;
// transition x from shared to private
privatize_barrier(numthreads,x,arraysize*sizeof(int));
```

(b) dynamic privatization using barriers

Figure 6: Example codes using Notary's privatization API

programming interface for privatization, which is described in more detail below.

Interfaces. If stack data is not shared amongst threads, the library/runtime statically marks all stack pages as private. Otherwise stack data is treated as equivalent to shared data, and handled similarly in the following discussion. Heap-allocated data complicates privatization because most compilers cannot statically guarantee whether the data will remain private or shared throughout its lifetime, particularly when pointers are explicitly used (e.g., C/C++). We avoid this problem by providing library functions for dynamic shared and private memory allocation (`shared_malloc()` and `private_malloc()`), and the freeing of those dynamic memory (`shared_free()` and `private_free()`). Figure 6(a) shows an example code snippet using these allocation routines.

In order to support dynamic privatization, Notary uses barriers to synchronize a group of threads on the sharing status of memory objects. This mechanism is inspired by Scott et al. from their work on privatization in Delaunay triangulation [34], and we extend it to allow the changing of an object's sharing status in both directions (from private to shared, and vice versa). These functions are `publicize_barrier()` and `privatize_barrier()`. In order to avoid potential deadlocks between threads, Notary prohibits these barriers from being used inside of transactions. Barrier functions take three arguments: the number of threads in the synchronization group to wait for before privatizing or publicizing the object, and the pointer and size of the object being made public or private. Figure 6(b) illustrates an example barrier usage.

Notary's page-based implementation. One possible implementation of Notary's privatization interfaces is to associate an object's sharing status with a page, and store the sharing status with address translation information. To implement `private_malloc()` and `shared_malloc()`, the library/runtime needs to actively separate private and shared objects with help from the OS. The OS enforces separation by allocating private and shared objects on separate virtual pages (and separate page frames), and marks shared pages with a privatization-specific "shared" bit. All objects on a specific page with shared or private status inherit that page's status, and a good policy for reducing the number of pages used is to co-locate objects with the same status on the same page. Any object with unknown status is considered shared for correct conflict detection. Metadata on objects and their sizes are stored in OS-accessible data structures.

The page-based bits can be associated with existing virtual address translation information and cached in hardware TLB structures. The HTM reads the shared bit during address translation (along with the current transactional mode of the thread context) to decide whether to isolate the memory address in the read- and write-sets (the read and write signatures for signature-based TM systems).

The barrier functions `privatize_barrier()` and `publicize_barrier()` can be implemented purely in software or, if available, with hardware barrier instructions. In the OS, these barriers translate to TLB-shootdowns, in which the sharing status for the page is set or unset. Multiple objects on the same page can be affected, and Notary can handle these cases with different levels of implementation complexity. When changing an object from private to shared, the OS can safely mark the entire page as shared and still maintain correct conflict detection. When changing an object from shared to private Notary's default action is to change the page's status only if there are no additional shared objects on the same page. In more complex implementations the OS throws an exception about the object being privatized. One way for the programmer to handle this exception is to free up this object's memory (with `shared_free()`), and re-allocate its memory using `private_malloc()`. If these actions cause a significant performance penalty, an alternative implementation may be needed. If fine-grained memory protection support is available (e.g., Mondrian [42]), the sharing status on individual memory words can change transparently to the programmer.

5. Platform & Methodology

5.1 Base CMP system

Our simulations model a 16-core CMP system. The in-order, single-issue cores each have 32kB, 4-way private writeback L1 I and D caches. A unified, 16-bank, 8MB, 8-way L2 cache is shared by all cores. A packet-switched interconnect with 64B links connects the cores and cache banks. Two on-chip memory controllers connect to standard DRAM banks. Cache-coherence is maintained via an on-chip directory which stores a bit vector of sharers and implements the MESI protocol. Due to better scalability [17], BIND runs on top of a 32-core CMP with the same memory system and interconnect parameters as above. The CMP implements the LogTM-SE [45] HTM system.

5.2 Workloads

We examine four sets of workloads: two micro-benchmarks (BTree and Sparse Matrix), two applications from SPLASH-2 [43] (Barnes and Raytrace), five applications from STAMP [28] (Vacation, Genome, Delaunay, Bayes, and Labyrinth), and one multi-threaded domain name service (DNS) server workload (BIND) [16]. BTree performs concurrent operations to a shared BTree data structure. Sparse Matrix is an algorithm taken from a dense column vector multiplication kernel. Barnes and Raytrace are chosen from SPLASH-2 because they exert the most signature pressure in that workload suite. STAMP represents workloads which have large transactions. The

Table 5: Workload Characteristics

Bench.	Xacts	Read-Set	Write-Set	Max Read-Set	Max Write-Set
BTree	100,000	13.4	1.3	21	16
Sparse Matrix	1,091,362	5.0	1.0	5.0	1.0
Barnes	2,364	6.1	4.6	40	40
Raytrace	47,765	5.3	2.0	596	3.0
Vacation	30,011	25	3.2	127	25
Genome	20,930	15	1.7	154	18
Delaunay	2,146	68	44	610	420
Bayes	550	73	37	2,061	1,597
Labyrinth	158	70	91	269	264
BIND	5,327	6.4	5.0	50	28

versions of Vacation and Genome used for our evaluations are modified from the original by converting coarse-grained transactions to fine-grained transactions. For Genome we use a smaller chunk size than the original. For Vacation we partition a client's travel request into smaller transactions. These changes improve workload scalability for smaller signature sizes. We also simulate a realistic transactional workload by converting lock-intensive portions of a lock-based DNS program, BIND 9.4.1 [16], to use transactions. Workload characteristics are shown in Table 5.

5.3 Simulation methodology

Performance results are collected using Simics full-system simulation [24] and Wisconsin GEMS [25]. Simics accurately models the SPARC architecture, and GEMS models the LogTM-SE HTM [45]. All workloads except BIND run on top of an unmodified Solaris 9 operating system, and BIND runs on top of OpenSolaris (Solaris 11). Each simulation is pseudo-randomly perturbed to produce error bars of 95% confidence on performance results.

5.4 Signature configurations

We simulate infinite-size, unimplementable "Perfect" signatures (with no false positives), and finite signatures of size 64b to 64kb (8B to 8kB). 64b signatures match the word size in current cores, and 64kb signatures match the performance of Perfect signatures for all workloads. LogTM-SE uses separate read and write signatures, with two hash functions for each signature. One hash takes even-numbered bits from the bit-fields, and the other odd-numbered bits. We simulate both PBX and H_3 signatures. PBX uses an 11-bit cache-index bit-field and a 10-bit non-overlapping PPN bit-field. H_3 uses a fixed pseudo-random Boolean matrix.

6. Evaluation Results

We first present execution time results of PBX versus H_3, and then the execution time results for Notary's privatization technique. All results are normalized to Perfect signatures.

6.1 Effectiveness of PBX versus H_3 hashing

Figure 7 shows execution times of PBX compared to H_3. For the ten workloads, the execution times for PBX and H_3 are similar, regardless of the signature size. Because they are different signature implementations, PBX may sometimes out-perform H_3 due to differences in signature utilization and conflict formation amongst concurrent transactions. In Labyrinth the differences arise from additional conflicts for H_3 in a latency-critical portion of a transaction, which copies a shared grid structure to a local copy of the grid. These conflicts are due to H_3 setting more signature bits than PBX in this transaction. This also occurs

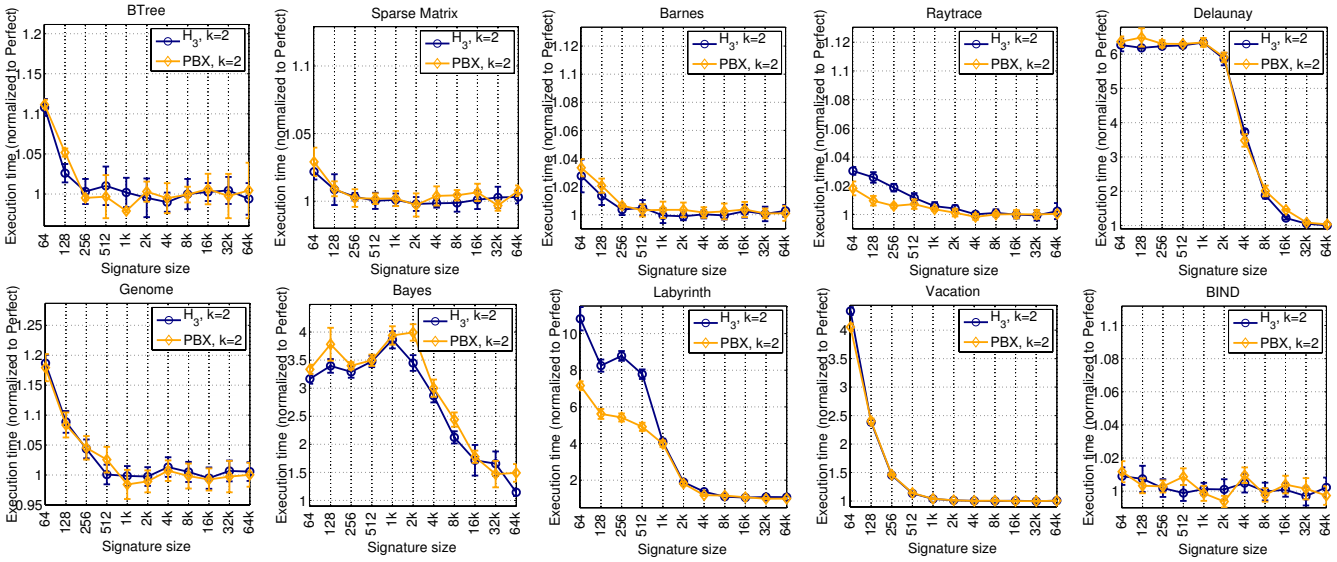

Figure 7: Execution time results of PBX versus H_3 hashing

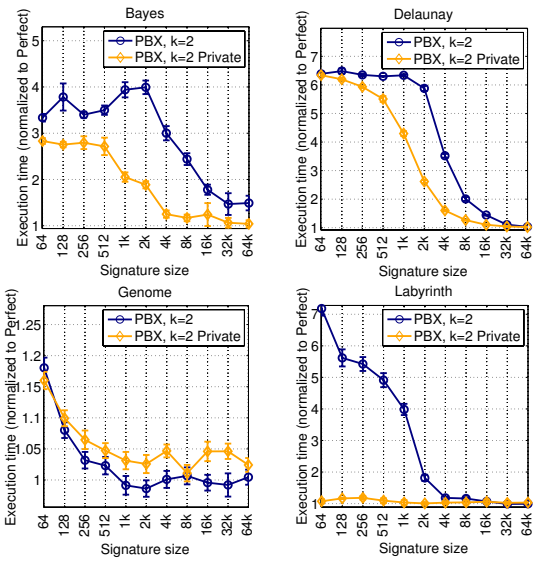

Figure 8: Privatization execution times

in Raytrace, in which H_3 sets more signature bits and leads to more non-transactional conflicts than PBX.

Implication 1: PBX achieves similar performance to H_3 hashing, but does so with much lower hardware cost.

6.2 Effectiveness of privatization

We simulate privatization in two stages. First, all stack references are marked private, as stack data are thread-private in our workloads. Privatizing the stack did not significantly reduce execution time for our workloads (results not shown). This is because the majority of memory references are to non-stack data.

Second, we simulate `private_malloc()` and `shared_malloc()` through separate heap regions that are pre-allocated at the beginning of the program. These regions service memory allocation requests. Isolation is dropped on the private heap region. We only show results for selected STAMP workloads. The remaining workloads either did not have a high transactional duty cycle or did not use any private heap data structures (and thus would not benefit from privatization). The results after both stages of privatization were completed are shown in Figure 8.

This figure illustrates several interesting trends. First, privatization reduces execution time. This allows smaller signatures to be used, which is attractive since in Section 3.6 we showed the power and area savings of smaller signatures. For example, in Bayes the execution time of 512b signatures after privatization is approximately equivalent to that of 8kb signatures before privatization (due to elimination of all false conflicts in the write signature). This is a reduction of 16x in the signature size needed! Second, these results parallel the results of our privatization micro-benchmark. Labyrinth is the workload that benefits most from privatization (approximately 86% reduction in normalized execution time from reducing false conflicts by 96%), and Table 6 reveals why this occurs. The largest reductions in transactional read- and write-set sizes occur in Labyrinth, with 88% and 94% reductions, respectively. Most of its reductions come from not isolating requests to per-thread private copies of the grid structure. In Genome, privatization is sometimes slower than the original because of an increased number of exceptions (e.g., register spill/fill, TLB misses).

Table 6: Read- & write-set sizes

Bench.	Before / After Privatization		% Reduction	
	Avg. read-set	Avg. write-set	Avg. read-set	Avg. write-set
Bayes	73 / 33	37 / 2.4	55	94
Delaunay	68 / 44	44 / 29	35	34
Genome	15 / 12	1.7 / 1.6	20	5.9
Labyrinth	70 / 8.5	91 / 5.1	88	94

Implication 2: Privatization is an effective technique to improve TM program execution time by reducing signature false positives.

Due to space limitations we do not present in-depth sensitivity analysis. Initial sensitivity analysis show similar results for signatures with four hash functions. Yen's forthcoming thesis [44] will present detailed analysis of additional signature configurations (e.g., with four hash functions) and graphs not shown in this paper.

6.3 Notary's applicability to other systems

PBX hashing may be implemented in any TM system which uses signatures (e.g., Bulk, LogTM-SE, SigTM) or in any signature-based non-TM systems (e.g., BulkSC, SoftSig, Atom-Aid, DeLorean, Rerun).

Notary's privatization support can be implemented on top of other TM systems. Privatization is attractive for software TMs which may incur high transactional bookkeeping overheads. It helps TM systems that store new values in place (by reducing abort overheads) and those that store new values in a separate buffer (by reducing commit overheads). For signature-based TMs, privatization can greatly reduce signature false conflicts. Signature-based non-TM systems can also use it. Programs which correctly classify and synchronize accesses to private data are not susceptible to memory races or memory consistency violations on that data.

Notary can also be used in conjunction with other techniques aimed at optimizing signature sizes. One such technique customizes signature sizes based on profile information of transaction read- and write-set sizes [28].

7. Conclusions & Future Work

This paper developed Notary, a coupling of two enhancements to hardware signature designs to tackle two problems: the high implementation overheads of H_3 hash

functions, and false positives due to signature bits set by private memory accesses. We introduced PBX hashing, which uses bit-entropy to carefully select random bit-fields and performs similar to H_3, but is implemented with much less hardware. Second, we proposed a privatization programming interface giving programmers the ability to declare which accesses cannot cause conflicts, and this can improve program execution time. Finally, Notary can be implemented in signature-based non-TM systems.

In the future, we plan on exploring how to dynamically select the bit-fields for PBX based on phase changes in entropy. This will allow signature-based TM and non-TM systems to have robust performance.

Acknowledgements

We thank Atif Hashimi for his guidance on Verilog simulations, Jayaram Bobba, Dan Gibson, Mike Marty, and Yasuko Watanabe for comments and proofreading. Finally we thank the Wisconsin Condor group, the UW CSL, and Virtutech for their assistance.

References

[1] M. Abadi, A. Birrell, T. Harris, J. Hsieh, and M. Isard. Dynamic Separation for Transactional Memory. Tech. Report MSR-TR-2008-43, Microsoft Research, 2008.

[2] A. R. Alameldeen, C. J. Mauer, M. Xu, P. J. Harper, M. M. K. Martin, D. J. Sorin, M. D. Hill, and D. A. Wood. Evaluating Non-deterministic Multi-threaded Commercial Workloads. In *Proc. of the 5th Workshop on Computer Architecture Evaluation Using Commercial Workloads*, pages 30–38, Feb. 2002.

[3] AMD Corporation. AMD Introduces the World's Most Advanced x86 Processor, Designed for the Demanding Datacenter. www.amd.com/us-en/Corporate/VirtualPressRoom/0,,51_104_543_15008~119768,00.html.

[4] W. Baek, C. C. Minh, M. Trautmann, C. Kozyrakis, and K. Olukotun. The OpenTM Transactional Application Programming Interface. In *PACT*, Sept. 2007.

[5] C. Ballapuram, K. Puttaswamy, G. H. Loh, and H.-H. S. Lee. Entropy-Based Low Power Data TLB Design. In *Proc. of the Intnl. Conf. on Compilers, Architecture, and Synthesis for Embedded Systems*, Oct. 2006.

[6] J. C. Becker, A. Park, and M. Farrens. An Analysis of the Information Content of Address Reference Streams. In *Micro-24*, Nov. 1991.

[7] B. H. Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Communications of the ACM*, 13(7):422–426, July 1970.

[8] J. L. Carter and M. N. Wegman. Universal Classes of Hash Functions (extended abstract). In *Proceedings of the 9th Annual ACM Symposium on Theory of Computing*, pages 106–112, 1977.

[9] L. Ceze, J. Tuck, C. Cascaval, and J. Torrellas. Bulk Disambiguation of Speculative Threads in Multiprocessors. In *ISCA-33*, June 2006.

[10] L. Ceze, J. Tuck, P. Montesinos, and J. Torrellas. BulkSC: Bulk Enforcement of Sequential Consistency. In *ISCA-34*, June 2007.

[11] D. Citron and L. Rudolph. Creating a Wider Bus Using Caching Techniques. In *HPCA-1*, pages 90–99, Feb. 1995.

[12] A. Gonzalez, M. Valero, N. Topham, and J. M. Parcerisa. Eliminating Cache Conflict Misses Through XOR-Based Placement Functions. In *Proc. of the 1997 Intnl. Conf. on Supercomputing*, July 1997.

[13] D. W. Hammerstrom and E. S. Davidson. Information Content of CPU Memory Referencing Behavior. In *ISCA-4*, Mar. 1977.

[14] M. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *ISCA-20*, May 1993.

[15] D. R. Hower and M. D. Hill. Rerun: Exploiting Episodes for Lightweight Race Recording. In *ISCA-35*, June 2008.

[16] Internet Systems Consortium. Berkeley Internet Name Domain (BIND). http://www.isc.org/index.pl?/sw/bind/.

[17] T. JINMEI and P. Vixie. Implementation and Evaluation of Moderate Parallelism in the BIND9 DNS Server. In *2006 USENIX Annual Technical Conference*, June 2006.

[18] M. Kharbutli, K. Irwin, Y. Solihin, and J. Lee. Using Prime Numbers for Cache Indexing to Eliminate Conflict Misses. In *HPCA-10*, 2004.

[19] M. Kharbutli, Y. Solihin, and J. Lee. Eliminating Conflict Misses Using Prime Number-Based Cache Indexing. *IEEE TOC*, 54(5), 2005.

[20] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-Way Multithreaded Sparc Processor. *IEEE Micro*, 25(2), Mar/Apr 2005.

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

[22] H. Le, W. Starke, J. Fields, F. O'Connell, D. Nguyen, B. Ronchetti, W. Sauer, E. Schwarz, and M. Vaden. IBM POWER6 microarchitecture. *IBM Journal of R&D*, 51(6), 2007.

[23] B. Lucia, J. Devietti, K. Strauss, and L. Ceze. Atom-Aid: Detecting and Surviving Atomicity Violations. In *ISCA-35*, June 2008.

[24] P. S. Magnusson et al. Simics: A Full System Simulation Platform. *IEEE Computer*, 35(2):50–58, Feb. 2002.

[25] M. M. K. 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*, Sept. 2005.

[26] A. Matveev, O. Shalev, and N. Shavit. Dynamic Identification of Shared Transactional Locations. Tech. report, Tel-Aviv Univ., 2007.

[27] M. Milovanovic, R. Ferrer, O. S. Unsal, A. Cristal, X. Martorell, E. Ayguade, J. Labarta, and M. Valero. Transactional Memory and OpenMP. In *Proceedings of the International Workshop on OpenMP*, June 2007.

[28] 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 *ISCA-34*, June 2007.

[29] P. Montesinos, L. Ceze, and J. Torrellas. DeLorean: Recording and Deterministically Replaying Shared-Memory Multiprocessor Execution Efficiently. In *ISCA-35*, June 2008.

[30] A. Park and M. Farrens. Address Compression Through Base Register Caching. In *23rd Annual Symposium and Workshop on Microprogramming and Microarchitectures*, Nov. 1990.

[31] M. Ramakrishna, E. Fu, and E. Bahcekapili. Efficient Hardware Hashing Functions for High Performance Computers. *IEEE Transactions on Computers*, 46(12):1378–1381, 1997.

[32] B. R. Rau. Pseudo-randomly interleaved memory. In *ISCA-18*, 1991.

[33] D. Sanchez, L. Yen, M. D. Hill, and K. Sankaralingam. Implementing Signatures for Transactional Memory. In *Micro-40*, Dec. 2007.

[34] M. L. Scott, M. F. Spear, L. Dalessandro, and V. J. Marathe. Delaunay Triangulation with Transactions and Barriers. In *IISWC*, pages 107–113, Sept. 2007.

[35] A. Seznec. A Case For Two-way Skewed-associative Caches. In *ISCA-20*, May 1993.

[36] C. E. Shannon. Prediction and Entropy of Printed English. *Bell Systems Technical Journal*, (30):50–64, 1951.

[37] M. F. Spear, V. J. Marathe, L. Dalessandro, and M. L. Scott. Privatization Techniques for Software Transactional Memory. Tech. Report 915, University of Rochester, Feb. 2007.

[38] D. Tarjan, S. Thoziyoor, and N. P. Jouppi. CACTI 4.0. Technical Report HPL-2006-86, Hewlett Packard Labs, June 2006.

[39] J. Tuck, W. Ahn, L. Ceze, and J. Torrellas. SoftSig: Software-Exposed Hardware Signatures for Code Analysis and Optimization. In *ASPLOS-13*, Mar. 2008.

[40] H. Vandierendonck and K. D. Bosschere. XOR-Based Hash Functions. *IEEE Transactions on Computers*, 54(7):800–812, 2005.

[41] 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, 1994.

[42] E. Witchel, J. Cates, and K. Asanovic. Mondrian memory protection. In *ASPLOS-10*, pages 304–316, Oct. 2002.

[43] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *ISCA-22*, pages 24–37, June 1995.

[44] L. Yen. *Signatures in Transactional Memory Systems*. PhD thesis, University of Wisconsin, Expected Dec. 2008.

[45] 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 *HPCA-13,* Feb. 2007.

[46] Z. Zhang, Z. Zhu, and X. Zhang. A Permutation-based Page Interleaving Scheme to Reduce Row-buffer Conflicts and Exploit Data Locality. In *Micro-33*, Dec. 2000.