Bloom Filter Guided Transaction Scheduling

Geoffrey Blake, Ronald G. Dreslinski, Trevor Mudge
Advanced Computer Architecture Laboratory
University of Michigan, Ann Arbor
{blakeg,rdreslin,tnm}@umich.edu

Abstract

Contention management is an important design component to a transactional memory system. Without effective contention management to ensure forward progress, a transactional memory system can experience live-lock, which is difficult to debug in parallel programs. Early work in contention management focused on heuristic managers that reacted to conflicts between transactions by picking the most appropriate transaction to abort. Reactive methods allow conflicts to happen repeatedly as they do not try to prevent future conflicts from happening. These shortcomings of reactive contention managers have led to proposals that approach contention management as a scheduling problem—proactive managers. Proactive techniques range from throttling execution in predicted periods of high contention to preventing groups of transactions running concurrently that are predicted likely to conflict.

We propose a novel transaction scheduling scheme called "Bloom Filter Guided Transaction Scheduling" (BFGTS), that uses a combination of simple hardware and Bloom filter heuristics to guide scheduling decisions and provide enhanced performance in high contention situations. We compare to two state-of-the-art transaction schedulers, "Adaptive Transaction Scheduling" and "Proactive Transaction Scheduling" and show that BFGTS attains up to a 4.6x and 1.7x improvement on high contention benchmarks respectively. Across all benchmarks it shows a 35% and 25% average performance improvement respectively.

1. Introduction

Transactional Memory (TM), first proposed by Herihly and Moss [15], has received attention from many researchers during the past decade as a replacement for locks in shared memory parallel programs. TM treats critical sections of code as atomic units: transactions either complete in totality or not at all. Programming with TM has been found simpler in user tests by Rossbach et al. [22]. In their studies the appearance of coarse grained locking semantics—placing transactions around critical sections to operate atomically—appears to be the main contributor to the ease of programmability. In addition, TM offers fine grained locking performance with fewer errors because programmers are not required to compose numerous locks in specific locking orders. This suggests TM can be an important tool for creating parallel programs efficiently.

These advantages of easier programming semantics and fine grained locking performance have led to numerous TM design proposals. TMs can be implemented in hardware, software or be a hybrid that uses hardware in the common case, and falls back to software when needed. Details covering these fundamental design decisions, as well as other design details such as memory versioning, conflict detection and isolation properties can be found in the book by Larus and Rajwar [17]. TM research has caught the attention of industry and some companies are developing transactional memory extensions to their architectures, such as AMD's Advanced Synchronization Facility [10], and Azul System's TM solution [11].

There are still open questions in TM, which include non-transactional operations such as I/O, reducing software transactional memory overheads, and effective contention management. For Hardware Transactional Memory Systems (HTMs), effective contention management is important as contention can lead to worse than serial performance that is difficult to debug.

The issues with contention were first explored in a TM performance pathologies paper by Bobba et al. [8]. Newer benchmarks representative of future TM applications, STAMP [9], use large, coarse-grained transactions exposing problems with contention not seen in earlier TM benchmarks like transactional versions of SPLASH2 [27]. SPLASH2 did not exhibit much contention because its scientific parallel programs used small, infrequent transactions. Early contention managers were primarily reactive using a backoff strategy. These reactive contention managers fixed up conflicts between transactions when they happened, but did not prevent future conflicts from occurring. Reactive contention managers are ideal for low con-

978-1-4244-9435-4/11/$26.00 ©2011 IEEE

tention situations, because they impose little runtime overhead. In high contention situations, reactive contention managers are ill suited as they can lead to worse than sequential performance. This has prompted researchers to look into transaction scheduling to avoid conflicts before they happen [2, 3, 4, 6, 12, 21, 26, 28]. These techniques all attempt to maximize parallel execution and minimize wasted work by trying to proactively avoid contention.

In this paper we propose "Bloom Filter Guided Transaction Scheduling" (BFGTS). It employs novel Bloom filter operations to characterize transaction behavior, using this information to schedule conflict free parallel execution. Our proposed technique also makes use of some small hardware structures to reduce runtime overheads. The key contributions are:

- Novel Bloom filter manipulations to infer transaction memory footprint behavior
- Hardware extensions to accelerate scheduling
- Compact, efficient software data structures

To evaluate our technique, we compare to two transaction scheduling techniques: "Proactive Transaction Scheduling" (PTS) [6] and "Adaptive Transaction Scheduling" (ATS) [28]. We find on average our techniques beat PTS by 25% (up to 1.7x on high contention benchmarks), and beats ATS by 35% (up to 4.6x on high contention benchmarks) when compared in identical configurations.

2. Background and related work

Contention managers have been receiving increased attention as they improve performance transparently to the programmer in TM systems. Work involving contention managers has been primarily done in Software TM (STM) as contention is very expensive in STM systems. Work done by Scherer and Scott [24, 25] was the first to go beyond simple backoff schemes developing multiple reactive backoff based contention managers that used many heuristics. They found that it is difficult to come up with a perfect reactive contention manager as no one set of heuristics from their studies was the clear winner. Work by Bai et al. [4] first proposed scheduling transactions to reduce contention. The main drawback to this work was that each benchmark needed a custom scheduling function to work. More general scheduling solutions have been proposed to allow for better scaling in the STM field. These include proposals by Ansari et al. [2, 3], Sonmez et al. [26], Dolev et al. [12], Dragojević et al. [13] and Maldonado et al. [18]. All involve scheduling transactions in some form, either by using a coarse grained metric such as commit rate, or attempting to determine pairs of transactions that should be serialized dynamically.

Contention management research for Hardware TM (HTM) has been less studied. Bobba et al. [8] identified many pathologies that can be encountered in HTMs and proposed some solutions, though these solutions were not investigated in depth. Work by Zilles et al. [29] is similar in principal to Ansari's "Steal on Abort" technique by stalling a transaction to disallow repeated conflicts. Dependence-Aware TM (DATM) proposed by Ramadan et al. [21] also looks to avoid contention but does so in a very different manner. The DATM technique tracks potential conflicts between memory locations and then forwards future values to allow these conflicting transactions to commit successfully without a rollback and restart. The main drawback to DATM is the complicated hardware it requires.

2.1. Adaptive transaction scheduling

ATS is a transaction scheduling technique proposed by Yoo and Lee [28]. ATS throttles concurrent transaction execution by monitoring per-transaction *conflict pressure* values which are represented by a moving average that increases on a conflict and decreases on commit. If the conflict pressure exceeds a preset threshold, transactions queue themselves onto a central wait queue to execute serially with respect to other transactions that have seen high contention. When the conflict pressure value decreases below the threshold, transactions bypass the wait queue and execute in parallel.

ATS is a simple scheduling proposal that requires little software to implement, and guarantees that if contention is very high it will gracefully degrade to the performance of a single lock for all critical sections. If contention is low, ATS has little impact on performance. The main drawback of ATS is that serializing all transactions to a central queue when contention rises above a threshold can be too pessimistic. ATS does not try to identify the transactions that could run concurrently without conflict. This can result in over-serialization.

2.2. Proactive transaction scheduling

PTS by Blake et al. [6], like ATS, attempts to schedule transactions to reduce contention and increase performance. Unlike ATS, it attempts to identify the transactions that can run concurrently with each other and serializes those that should not. PTS accomplishes this by profiling the pattern of conflicts between transactions during runtime. As conflicts happen, PTS fills in a global graph data structure (conflict graph) with the nodes representing transactions and the edges representing the confidence a conflict will occur between transactions. Before each transaction begins execution, PTS consults a table representing the currently running transactions in the system and then consults the conflict graph to arrive at a prediction of whether it should serialize against a running transaction or proceed. When a transac-

tion finishes its execution, it updates the graph, strengthening or weakening confidences between nodes. PTS does this by tracking which transactions it predicted to serialize behind, and on commit it uses a Bloom filter representing its read/write set and intersects it with the saved Bloom filters of the transactions it serialized behind. If the intersections are not null, indicating a conflict would have happened if the transactions executed concurrently, the confidence between nodes is strengthened, otherwise it is weakened to promote parallel execution. Work by Dragojević is very similar to PTS, but for software transactional memory systems where scheduling overheads are less important.

PTS is both more complicated and more expensive in terms of software overhead than ATS. However, because it identifies groups of conflicting transactions it can schedule more optimistically than ATS. PTS performance is still suboptimal because of: 1) very large software graph structure that can be 10's of megabytes in size; 2) overhead of executing a scan of software structures on every transaction begin; and 3) rudimentary Bloom filter use.

3. Motivation

3.1. Transaction behavior

The dynamic nature of code executed inside a transaction makes it hard to predict good schedules that avoid conflicts. Just tracking conflict history or contention rate is not enough to get a full picture of the program's behavior. To form good scheduling decisions a proactive scheduler needs to be able to identify the behavior of transactions and how they are being affected by conflicts. This can help guide a scheduler to be more optimistic scheduling some transactions while scheduling pessimistic for others.

Take the following synthetic example. Assume a group of transactions are modifying locations in memory. Some transactions continually modify the same general locations in memory each time they are executed, shown in Figure 1(a). This transaction exhibits a high amount of memory locality on each consecutive execution. For the rest of this paper we will term this locality property "*Similarity*". Other transactions may jump around, working in different regions of memory each time they execute, shown in Figure 1(b). This transaction exhibits low similarity on consecutive executions. In terms of transaction conflicts, if two transactions having low similarity conflict in the past, this conflict is likely to be transient, e.g., inserting to a hash table. Conversely, if two transactions conflict and they have high similarity, this conflict is likely to persist, e.g., enqueuing and dequeuing from a queue. This property can help a scheduler identify such behaviors and treat conflicts accordingly. We define similarity as a value between 0 and 1 as follows:

$$Similarity = \frac{SetSize(RWSet_{t-1} \cap RWSet_t)}{AvgRWSetSize} \quad (1)$$

Where $SetSize$ counts the number of entries in a set, $AvgRWSetSize$ is the historical average number of entries in the transaction's read/write set, and $RWSet$ is the set of addresses touched by the transaction. Similarity in this case is calculated using the just completed transaction from time t and the previous execution $t-1$. The more entries in common between two $RWSet's$, the higher the similarity (closer to 1). Looking at Figure 1 (a), the similarity for $Tx1$ would be close to 1 as each consecutive execution touches similar memory.

This type of behavior was measured in the STAMP benchmarks. Table 1 shows the conflict graph for each transaction, and the measured value of each transaction's *similarity*. The *Conflict Graph* in Table 1 is a matrix representation of the conflict graph seen by transactions during the execution of the STAMP benchmarks. Each number in the column *Conflict Graph* represents a conflict occurred at some point between that transaction ID and the transaction ID in listed in column *Tx*. Each transaction ID represents a transaction defined in the code. For the Delaunay benchmark, it has transactions that conflict with every other transaction in the system. The transactions `0`, `2` and `3` have a high similarity and should be serialized, while the transaction `1` has a very low similarity and should be treated by a scheduler as a transaction that has transient conflicts. A scheduler that can better identify transaction behavior will be able to make more informed scheduling decisions.

3.2. Using bloom filters to extract transaction behavior

As seen in the previous section, calculating similarity requires a set intersection, which can be expensive if the sets are compared pairwise. This is not feasible in a HTM, so we use Bloom Filters [7] to represent and work on transaction read/write sets efficiently. As shown by Sanchez et al. [23] implementing bloom filters in hardware can be done efficiently in hardware. A unique contribution of this work is to develop Bloom filter manipulations to estimate similarity.

We use work by Michael et al. [19] to develop the Bloom filter operations to estimate similarity. The Bloom filter manipulations were originally developed to for fast join operations in large distributed databases. The main equations used are the set size estimations(denoted as $S^{-1}(t)$) of encoded Bloom filters(denoted as $S(t)$). Equation 2 calculates the set size estimation of an encoded Bloom filter where t is the number of bits set, m is the total size in bits of the Bloom filter, and k is the number of hash functions used.

$$S^{-1}(t) \cong \frac{\ln(1-\frac{t}{m})}{k * \ln(1-\frac{1}{m})} \quad (2)$$

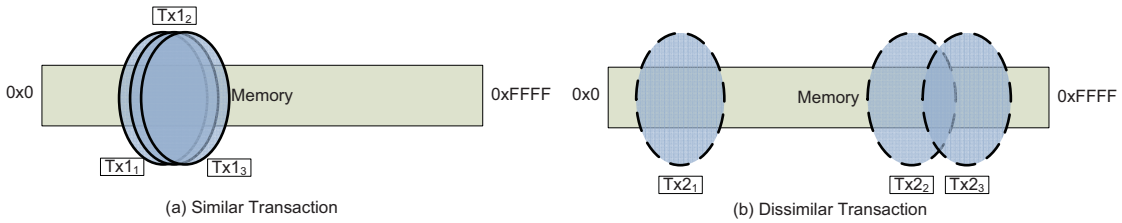


Figure 1. Example transaction executions that show the difference between similar execution behaviors (a) and dissimilar execution behaviors (b) over time.

Benchmark	Tx	Conflict Graph	Similarity	Benchmark	Tx	Conflict Graph	Similarity
Delaunay [16]	0:	0 1 2	0.64	Intruder	0:	0	0.67
	1:	0 1 2 3	0.04		1:	1 2	0.40
	2:	0 1 2 3	0.56		2:	1 2	0.66
	3:	1 2 3	0.90				
Genome	0:	0	0.12	Ssca2	0:	0	0.90
	1:		0.25		1:		0.90
	2:	2 3	0.65		2:	2	0.57
	3:	2	0.74				
	4:	4	0.29				
Kmeans	0:	0	0.38	Labyrinth	0:	0	0.86
	1:	1 2	0.67		1:	1 2	0.45
	2:	1	0.68		2:	1 2	0.90
Vacation	0:	0	0.26				

Table 1. Matrix representation of the conflict graph observed during the execution of each STAMP benchmark and measured similarity for each unique transaction.

An estimation of set size of the intersection between two Bloom filters shown in Equation 3 is also used.

$$S^{-1}_{AND}(t) \cong S^{-1}_1(t) + S^{-1}_2(t) - S^{-1}(S_1(t) \cup S_2(t)) \quad (3)$$

Finally, Equation 3 derives the *"Similarity"* metric using Bloom filters to represent read/write sets.

$$Similarity \cong \frac{S^{-1}_{AND}(t)}{AvgRWSetSize(Tx_n)} \quad (4)$$

4. Implementation

This section describes the hardware and software implementation of BFGTS. The design uses fine-grained scheduling between transactions and borrows concepts from PTS. BFGTS maintains a graph structure in software of nodes and edges to represent conflict history and confidence to facilitate scheduling decisions. The majority of BFGTS is implemented as a software runtime that sits between the application and the Operating System. A small TLB like hardware accelerator is also present that operates when it sees a `TM_BEGIN` instruction from the processor.

During the discussion of BFGTS there are two types of transaction IDs (TxID) that will be used in this section: *"Static Transaction ID"*(sTxID) and *"Dynamic Transaction ID"*(dTxID). An sTxID is statically assigned in the program code. A dTxID is a concatenation of thread ID and sTxID.

4.1. Scheduling hardware accelerator

In BFGTS, before a transaction begins execution it must scan a global array called the *CPU Table*—a list of the transactions running on the processors in the system. At each entry in the table, a confidence value representing the likelihood of a conflict between the transaction to be scheduled and the running transaction is retrieved from a global graph data structure. If the confidence exceeds a threshold value the transaction is serialized. Scanning the *CPU Table* at the start of every transaction adds overhead to each transaction.

BFGTS minimizes this overhead by using a hardware accelerator to scan the *CPU Table*, look up confidence values, and compare them to a preset threshold. This is then used to effect a scheduling decision in a few cycles. This accelerator is triggered upon seeing a `TX_BEGIN` instruction. These operations are relatively simple, therefore the hardware is small. The hardware implements the algorithm shown in Example 1.

The scheduling hardware is illustrated in Figure 2. It consists of a small cache, a handful of control registers, and logic connected to the coherent interconnect. Each proces-

Example 1 Lookup algorithm implemented by hardware accelerator

```
1  bool scheduleTransaction(int sTxID)
2  {
3   for (i=0; i < sizeof(CPUTable); i++){
4    confidx=CPUTable[i]>>shift_value;
5    conf=confidenceTable[sTxID][confidx];
6    if (conf > threshold) {
7     dTxID_wait_on=CPUTable[i];
8     return true;//conflict predicted
9    }
10  }
11  return false;//no conflict predicted
12 }
```

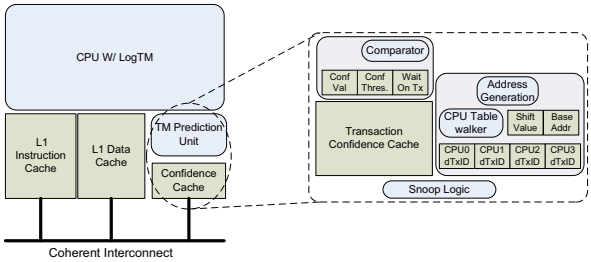


Figure 2. Hardware required to accelerate scheduling on TX_BEGIN for a 4 core system.

sor gets an identical predictor unit so the predictions are fully distributed. The control registers consist of a CPU Table that represents all the remote processors in the system and the dTxID of its currently executing transaction. The other registers are as follows: a physical base address of the confidence value table to index into, the confidence threshold to compare confidence values against, a shift register for truncating dTxIDs in the CPU Table to sTxIDs, and a register to hold the dTxID of a transaction to serialize against for later access by software. The hardware predictor contains a small cache that is exclusively used for caching the confidence table. The cache is necessary because the confidence tables can be pushed out of the L1 caches, increasing the time it takes to make a prediction. The cache is also modified to fetch cache lines evicted by an invalidate snoop. This is required to prevent always taking a miss when accessing the cache because the main processor writes to the confidence tables frequently. The hardware overhead of the small cache and accelerator is very small.

To interface the software with the hardware prediction unit, the `TX_BEGIN` instruction is modified to trigger the predictor to form a prediction, and a new instruction `TX_QUERY_PREDICTOR` is added to modify the control registers of the predictor. `TX_BEGIN` traditionally puts the CPU into transactional mode, takes a register checkpoint, but takes no register arguments. `TX_BEGIN` now takes a vector to a `suspendTx()` function for the processor to jump to if the hardware predictor returns that a conflict is likely and the transaction should serialize. `TX_BEGIN` triggers the hardware predictor to perform the algorithm in Example 1 and waits for it to return either yes, a conflict is likely and jump to `suspendTx()` or continue execution. The `TX_QUERY_PREDICTOR` instruction acts like the `ioctl()` system call for the accelerator engine. The instruction is used to communicate information such as the physical address of the confidence table to use in the hardware predictor, query what dTxID to serialize against, set the confidence threshold to use, and query if a dTxID is still executing in the system to allow busy waiting.

Additional requests are added to the coherent interconnect to allow the predictors to update their arrays representing the state of each remote CPU. When a transaction is allowed to execute, it broadcasts onto the interconnect the dTxID of the starting transaction as well as the CPU ID. The other predictors snoop this broadcast and update their arrays accordingly. This is similar to TLB shootdown mechanisms when page table structures are updated on one CPU that need to be updated to other CPU's TLBs. On a transaction commit or abort, the CPU broadcasts the CPU ID along with the transaction outcome for the other predictors to update their internal state.

4.2. Software runtime component

The rest of BFGTS is kept in a software runtime to do book keeping operations, such as updating the confidences and calculating similarities of transactions. These book keeping functions can be quite complicated. Therefore a hardware mechanism would be infeasible as the amount of logic and storage required would be on the order of an additional processor core. Properly optimized software with the necessary ISA support is sufficient.

4.2.1. Data structure organization. The data structures used in BFGTS are inspired by PTS, but modified to be more efficient in both layout and space. An overview of the data structures are shown in Figure 3.

The first data structure is a set of confidence tables that are allocated per processor. This allows easy caching in the private cache attached to the hardware accelerator. The confidence tables hold the values that predict how likely a conflict is between two transactions if they were to execute concurrently in the future. In PTS, the confidence table was one global table that had a confidence entry for each dTxID. This table could grow to be 10's of MBs in size. Instead of tracking a confidence for every pair of dTxIDs, BFGTS compresses the table to only maintain confidence values between each pair of sTxIDs assigned in the code. By tracking

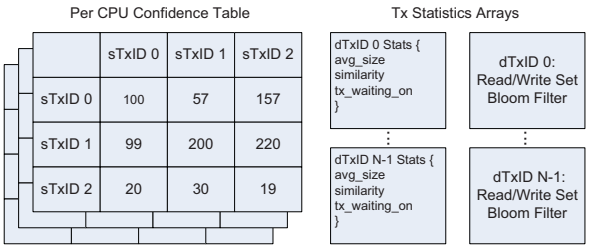


Figure 3. Data-structures for the confidence tables, transaction statistics table and Bloom filter tables kept in virtual memory.

only sTxIDs, the confidence table reduces to a maximum size of *800Bytes* for the benchmarks tested.

The second data structure required is an array of statistics kept for each dTxID that is encountered during runtime. For each dTxID three items are stored: average transaction size, similarity, and if a conflict was predicted, the dTxID of the conflicting transaction. The final data structure is a table of the most recent Bloom filters for each dTxID. The Bloom filters are used to calculate the average similarity of each dTxID, and to update confidence of conflict between sTxIDs on commit.

These data structures grow in similar fashion to the data structures of PTS. The *Confidence Tables* grow in memory in $O(M^2)$ where M is the number of transactions declared statically in the code. The *Tx Statistics Array* grow in memory in $O(NM)$ where N is the number of threads and M is the number of transaction declared in the code. These structures can grow to be unbounded, and therefore may be infeasible for very large transactional codes. A solution to this may be to allow aliasing in the prediction data structures—multiple transactions mapping to the same *Confidence Table* and *Tx Statistics Array* locations. This is left as future work and not explored in this paper.

4.2.2. Scheduling subroutines. The bulk of BFGTS exists as a software runtime. The software executes in user space, and is fully distributed. Three scheduling operations are done in software: Transaction Suspend, Transaction Abort, and Transaction Commit.

Transaction Suspend is the routine that the CPU vectors to when the `TX_BEGIN` instruction is informed by the hardware predictor a conflict is likely and the transaction needs to serialize. Example 2 illustrates how predicted conflicts are serialized in BFGTS. The dTxID of the transaction being serialized against is recorded for use later during transaction commit. If a $dTxID_i$ is predicted to conflict with another $dTxID_j$ that is historically small, then $dTxID_i$ stalls waiting for $dTxID_j$ to commit or abort. If $dTxID_j$ is larger than a small transaction threshold then it is suspended and another thread switched in. In

Example 2 Suspend Transaction Handling Pseudo Code

```
1  void suspendTx(int dTxID, int dTxIDSusp)
2  {
3   sim=0.5*(simOf(dTxID)+simOf(dTxIDSusp));
4   decay=decayVal*(1-sim);
5   decConfProb(sTxID,sTxIDSusp,decay);
6   statsTable[dTxID].txWaitingOn=dTxIDSusp;
7   if(avgTxSize(dTxIDSusp)>=SMALL_TX_SIZE){
8    pthread_yield();
9   } else {
10   stallOnTx(dTxIDSusp);
11  }
12  restore_checkpoint();
13 }
```

Example 3 Conflict Handling Pseudo Code

```
1 void txConflict(int dTxId,int dTxIdConf)
2 {
3  sim=0.5*(simOf(dTxId)+simOf(dTxIdConf));
4  inc=incVal*sim;
5  incConflictProb(dTxID,dTxIDConf,inc);
6  incConflictProb(dTxIDConf,dTxID,inc);
7 }
```

BFGTS `pthread_yield()` is used to switch threads. Upon exiting `suspendTx()` the transaction restores its register checkpoint and jumps to the PC to re-execute the `TX_BEGIN` instruction. To allow transactions to return to scheduling optimistically, a *decay* operation is used to slowly decrease the confidence that a conflict will occur between two sTxIDs. Decay is weighted by the average similarity of the two dTxIDs that are predicted to conflict to drive how quickly decay occur. If a conflict is predicted between two transactions and they are both very similar to themselves, then a predicted conflict is likely to be accurate, and the decay is small. On the other hand, if the transactions are dissimilar, the decay will be large, allowing the confidence to decay quickly to allow the two transactions to be scheduled concurrently.

On *Transaction Abort* due to a conflict, first the transaction rolls back its speculatively written state. Then it calls the `txConflict()` routine presented in Example 3 to increment confidence values of future conflict between the two dTxIDs. Again similarity is used to guide how much the confidence is incremented by.

When a transaction commits, various book keeping for that transaction needs to happen for accurate scheduling in the future. These operations are shown in pseudo-code in Example 4. These items are the average transaction size, the confidence between dTxIDs if one serialized against the other, and the average similarity of the committed transac-

Example 4 Pseudo code for routines used during Transaction Commit.

```
1  void commitTx(int dTxID)
2  {
3   updateAvgSize(dTxID);
4   updateBloom(dTxID);
5   int waitingOn=
6     checkWasSerialized(dTxID);
7
8   if (waitingOn!=NO_TX){
9    sim=0.5*(simOf(dTxID)+simOf(waitingOn));
10   if(intersectBlooms(dTxID,waitingOn)){
11    incConfProb(dTxID,waitingOn,
12                incVal*sim);
13   } else {
14    decConfProb(dTxID,waitingOn,
15                decVal*(1-sim));
16   }
17  }
18 }
19
20 void updateBloom(int dTxID)
21 {
22  nBloom=readCPUBloomFilter();
23  uBloom=UNION(nBloom,
24               bloomFilterTable[dTxID]);
25  newSim=calcSim(nBloom,
26                 bloomFilterTable[dTxID],
27                 uBloom);
28  newSim=
29   newSim/txStats[dTxID].avgTxSize;
30  txStats[dTxID].sim=
31   0.5*(txStats[dTxID].sim+newSim);
32 }
33
34 double calcSim(nBloom,oBloom,uBloom)
35 {
36  den=NUMHASHBITS*ln(1-1/NUMBLOOMBITS);
37  newSize=
38   ln(1-(bitCnt(nBloom)/NUMBLOOMBITS))/den;
39  oldSize=
40   ln(1-(bitCnt(oBloom)/NUMBLOOMBITS))/den;
41  unionSize=
42   ln(1-(bitCnt(uBloom)/NUMBLOOMBITS))/den;
43  return (newSize+oldSize-unionSize);
44 }
```

tion. To update the confidence of a conflict occurring in the future between two transactions that serialized the respective Bloom filters are intersected. If an intersection is not null then the confidence is incremented, otherwise it is decremented weighted by similarity.

Updating similarity is the most expensive part of BFGTS. As seen in pseudo-code in Example 4, calculating similarity requires two expensive functions: `bitCnt()`, and `ln()`. However, modern ISAs support both operations at the instruction level. A low latency 64-bit wide population count instruction, and a floating point logarithm instruction exist in modern ISAs like x86. These instructions are: `popcnt` and `fyl2x` [1]. The latencies of these instructions are 2-cycles and 13-cycles respectively for the AMD K10 architecture [14].

The transaction commit stage of the scheduling runtime can be particularly expensive, especially for small transactions, adding 100's of cycles of overhead to a transaction that may only be a few 10's of cycles in length. To reduce the overhead for small transactions similarity is updated for these transactions once every n commits. Large transactions are able to amortize the overhead of updating similarity on every commit, and usually benefit from the added scheduling accuracy.

4.3. BFGTS-HW/Backoff algorithm

To further reduce overhead, we present a hybrid BFGTS predictor borrowing ideas from Yoo and Lee's [28] ATS to allow the runtime to switch between using randomized backoff when contention is low and BFGTS using the hardware accelerator when contention is high. To measure contention ATS's metric *conflict pressure* is used to determine when to switch between BFGTS and randomized backoff with the goal of saving execution overhead. To implement the HW/Backoff predictor small changes are made to the presented BFGTS algorithm and described in the following paragraphs.

On `TM_BEGIN` the runtime checks the conflict pressure for the *sTxID* that wishes to execute, if it is over a set threshold then BFGTS is enabled and a scheduling prediction is made to suspend or continue execution. Otherwise, no prediction is made and the transaction begins execution. This allows the BFGTS-HW/Backoff predictor to save overhead on a transaction begins by not always having to walk the *CPU Table* on `TX_BEGIN`.

When a transaction commits, it checks *conflict pressure* first in `commitTx()` from Example 4 to determine if the transaction needs to perform the Bloom filter calculations. When *conflict pressure* is low, `commitTx()` skips performing the similarity calculations eliminating scheduling overhead. To update *conflict pressure*, the BFGTS-HW/Backoff algorithm increases pressure on aborts in the function `txConflict()` from Example 3, and predicted conflicts in `suspendTx()` from Example 2 in the same fashion as ATS. On commits, BFGTS-HW/Backoff decreases *conflict pressure*. Section 5 will show that being able to switch between BFGTS and randomized backoff eliminates enough overhead to allow larger Bloom filters to be used and in some cases increase performance.

Feature	Description
Processors	16 one IPC Alpha cores @ 2GHz
Special Instructions	popcnt 2-cycle latency, fyl2x 15-cycle latency
L1 Caches	64kB, 1 cycle latency, 2-way associative, 64-byte line size
Tx Confidence Cache	2kB, 16-way associative 1 cycle latency, 64-byte line size
L2 Cache	32MB, 32 cycle latency, 16-way associative, 64-byte line size
Interconnect	Shared bus at 2GHz
Main Memory	2048MB, 100 cycles latency
Linux Kernel	Modified v2.6.18
Contention Managers	PTS, ATS, BFGTS-SW, BFGTS-HW, BFGTS-HW/Backoff, BFGTS-NoOverhead
Signature Size	512bit-8192bit for BFGTS commit routines, perfect signature used for conflict detection.

Table 2. M5 Simulation Parameters

Benchmark	Input Parameters
Delaunay [16]	-i large.2 -m30 -t64
Genome	-g4096 -s32 -n524288 -t64
Kmeans	-m20 -n20 -t0.05 -i random50000_12 -p64
Vacation	-n8 -q10 -u80 -r65536 -t131072 -c64
Intruder	-a10 -l32 -n8192 -s1 -t64
Ssca2	-s15 -i1.0 -u1.0 -l3 -p3 -t64
Labyrinth	-i random-x96-y96-z3-n128.tx -t64

*We chose not to present the Bayes benchmark because of its non-deterministic finishing conditions as noted in [9], which makes direct comparisons between contention managers inconclusive

Table 3. STAMP Benchmark input parameters

5. Evaluation

5.1. Methodology

The M5 Full System Simulator [5] is used to evaluate BFGTS. The baseline TM system is based on LogTM [20] and has Operating System (OS) support. Three different scheduling based contention managers are evaluated: Adaptive Transaction Scheduling, Proactive Transaction Scheduling, and Bloom Filter Guided Transaction Scheduling. The simulation parameters are detailed in Table 2. The latencies for the `popcnt` and `fyl2x` instructions are modeled as well. The hardware accelerator with accompanying *Tx Confidence Cache* size as described in Table 2 has an area overhead of $\sim$**3%** of one 64kB L1 data cache.

The experiments for ATS, PTS and the four BFGTS variants assume an overcommitted system with 64 threads with four threads assigned per processor. We chose this configuration, as an overcommitted system is typical for systems running an OS. The advantage of such overcommitted systems is that when a thread blocks the OS can switch in another thread. This avoids leaving a core idle thus increasing throughput. We test the dynamically tuning software version of ATS developed by Yoo and Lee [28] using pthreads to suspend and wake threads when throttling. We test the standard version of PTS presented by Blake et al. [6]. We tested four versions of BFGTS: The hardware accelerated version presented in the previous sections BFGTS-HW, an all software version called BFGTS-SW, the hybrid BFGTS algorithm from Section 4.3 combining BFGTS-HW and Backoff managers called BFGTS-HW/Backoff, and BFGTS-NoOverhead. BFGTS-NoOverhead, as its name implies, implements BFGTS where all the software functions presented in Section 4.2 complete in one cycle. This is done to evaluate how well BFGTS predicts and schedules around conflicts when it does not have to amortize the cost of book keeping operations. BFGTS-NoOverhead also uses perfect read/write set signatures.

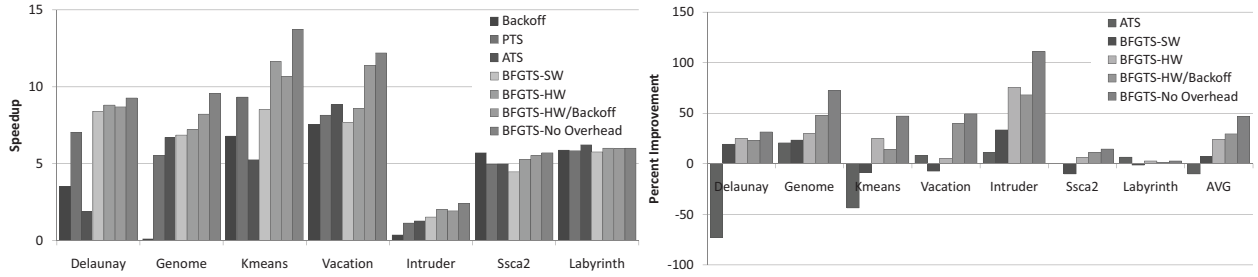
The transaction schedulers are evaluated with the STAMP benchmark suite [9]. The benchmark parameters are shown in Table 3. These benchmarks stress the TM system, especially the contention manager as they can suffer high contention when using a simple backoff manager as shown in Table 4. Statistics are collected only during the parallel phase of each benchmark. For the *Labyrinth* benchmark, we modify the code to perform the grid copy outside of the transaction as has been done by others. This allows some parallel scaling as unmodified it operates serially.

5.2. Results

5.2.1. Overall performance. The results presented in this section are the configurations of the BFGTS predictors with their optimal size Bloom filter. The Bloom filter size was varied from 512bits to 8192bits and Section 5.3.1 looks at performance sensitivity to Bloom filter size. Update frequency for small transactions was set to every 20 commits, and small transactions were defined as transactions with an average size of 10 cache lines or less. BFGTS-HW/Backoff uses contention pressure threshold value of 0.25 and heavily biases past history, therefore the frequency of switching between backoff and BFGTS-HW is slow.

Figure 4(a) shows the speedups attained for each contention manager (Backoff, ATS, PTS, BFGTS-SW, BFGTS-HW, BFGTS-HW/Backoff and BFGTS-NoOverhead) on a 16 CPU system over a single core. Figure 4(b) normalizes the speedup values to percent improvement over PTS. Table 4 shows the contention experienced by each tested contention manager.

Backoff is the worst performing contention manager. It experiences the highest contention, as can be seen in Table 4. This in turn leads to its poor performance compared to the proactive schedulers for the STAMP benchmarks as



(a) Speedup over one core

(b) Percent Improvement over PTS

Figure 4. Overall speedup and percent improvement over PTS for tested contention managers on a 16 processor system.

Benchmark	Backoff	PTS	ATS	BFGTS -SW	BFGTS -HW	BFGTS -HW/Backoff	BFGTS -NoOverhead
Delaunay	73.5%	28.0%	23.9%	22.2%	21.7%	23.7%	23.6%
Genome	61.1%	1.4%	1.0%	1.1%	1.1%	3.3%	3.8%
Kmeans	20.5%	2.9%	0.7%	1.2%	1.9%	6.5%	1.8%
Vacation	10.2%	8.9%	3.2%	4.6%	3.2%	5.3%	1.8%
Intruder	70.4%	4.6%	4.1%	5.5%	3.2%	9.9%	6.0%
Ssca2	0.1%	0.1%	0.1%	0.1%	0.1%	0.1%	0.1%
Labyrinth	20.2%	20.5%	6.5%	12.5%	9.3%	9.4%	9.3%

Table 4. Contention rates for the STAMP benchmarks for Backoff, PTS, ATS, BFGTS-SW, BFGTS-HW, BFGTS-HW/Backoff and BFGTS-NoOverhead on a 16 processor system.

seen in Figure 4. *Ssca2* is the exception because it experiences little contention and favors a very low overhead contention manager.

ATS does well for most benchmarks, it is only 10% worse than PTS, and is simpler in design. Most of the performance loss is on *Delaunay*, *Kmeans* and *Intruder* benchmarks. They have high contention which causes ATS to schedule pessimistically. This is primarily due to these benchmarks having dense conflict patterns and varying degrees of similarity as seen in Table 1. ATS is good for benchmarks with sparse conflict patterns, such as *Genome* or *Vacation*.

PTS is a more optimistic scheduler than ATS and therefore attains better performance. Because of its scheduling algorithm, PTS is both too optimistic and pessimistic at times as seen by the varied contention numbers in Table 4. This leads to poorer overall performance on average, and PTS loses to all BFGTS variants.

BFGTS-NoOverhead presents the best case performance of the BFGTS technique by modeling all scheduling operations completing in one cycle. As can be seen in Figures 4(a) and 4(b), BFGTS-NoOverhead is comparable or better than all the tested contention managers. It is on average 50% better than PTS. This is due to both its low overhead as well as better predictions from using perfect read-/write signatures for similarity calculations

BFGTS-SW uses no hardware acceleration, and does all scheduling operations in software. This adds overhead and decreases overall performance even though it has low contention. This overhead causes BFGTS-SW to lose on 4 out of 7 benchmarks to PTS, ATS or Backoff. These benchmarks are *Kmeans*, *Vacation*, *Ssca2* and *Labyrinth*. The remaining benchmarks see improvement, implying BFGTS-SW makes better predictions that either ATS or PTS, overcoming its software overheads. BFGTS-SW has only a 7% average improvement over PTS because of these overheads.

BFGTS-HW eliminate some of the overheads of BFGTS-SW by using the hardware accelerator from Section 4.1 to perform fast scheduling predictions. BFGTS-HW gets 25% better performance on average than PTS and 18% over BFGTS-SW. The hardware accelerator enables significant performance improvement. BFGTS-HW gets a maximum performance increase of 75% over PTS in the *Intruder* benchmark and 4.6x over ATS in *Delaunay*. BFGTS-HW does lose in 3 out of 7 benchmarks to ATS or Backoff because it also deals with overheads involved with calculating similarities on commit. Those benchmarks are *Vacation*, *Ssca2* and *Labyrinth*.

BFGTS-HW/Backoff further eliminates overhead by switching between backoff and BFGTS-HW getting 30% better performance than PTS on average. Switching between low overhead backoff and BFGTS-HW gets significant performance increases for the *Genome* and *Vacation* benchmarks, approaching BFGTS-No Overhead performance. This is because BFGTS-HW/Backoff is able save scheduling overhead for *Genome*. For *Vacation* BFGTS-

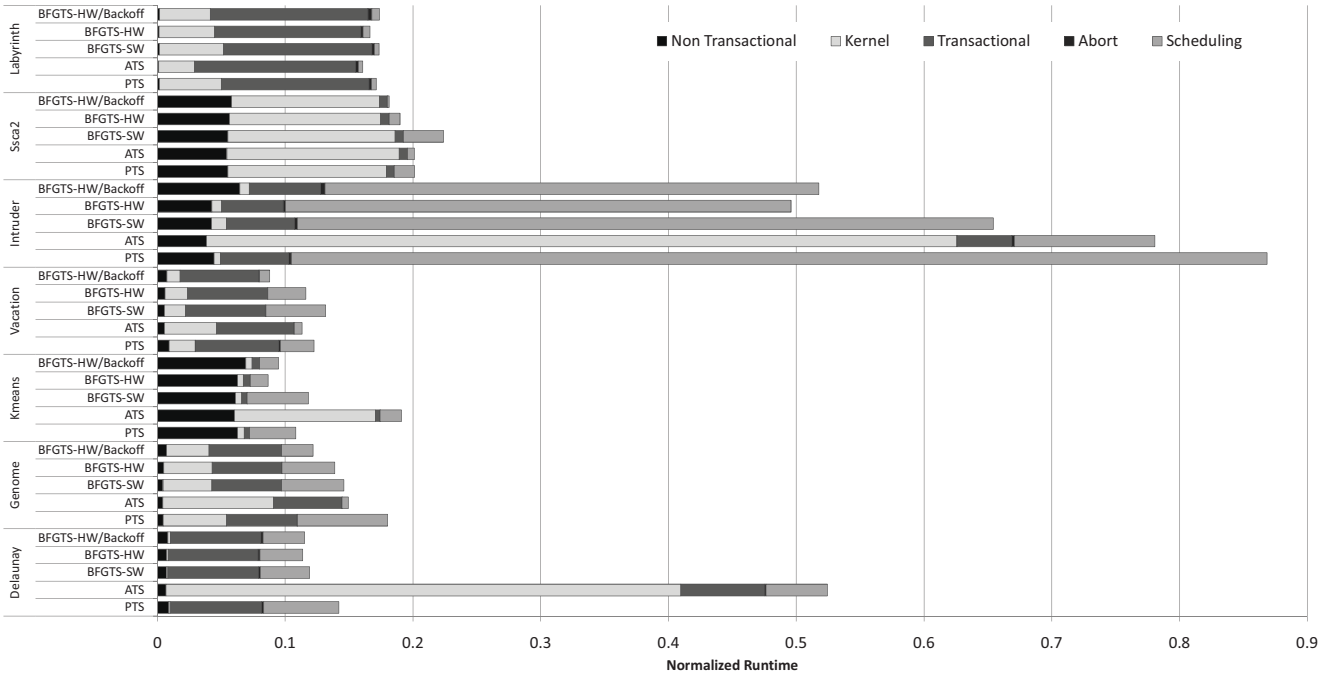


Figure 5. Time breakdown of the overheads for PTS, ATS, BFGTS-SW, BFGTS-HW and BFGTS-HW/Backoff for a 16 processor system.

HW/Backoff both saves overhead and can use larger Bloom filters for similarity calculations to make better predictions. It is also able to get closer to Backoff on *Ssca2* and *Labyrinth*. BFGTS-HW/Backoff does increase contention some, as seen in Table 4, because it is switching continuously between backoff and BFGTS-HW. This leads to slightly worse performance than BFGTS-HW for some benchmarks.

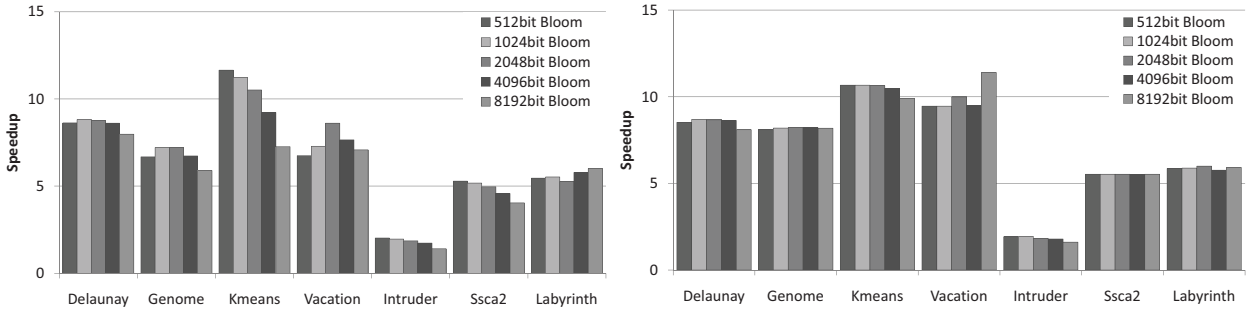
Figure 5 provides a breakdown of where each scheduling technique spends it execution time normalized to single processor execution time. In cases where ATS has worse performance—*Delaunay*, *Kmeans* and *Intruder* benchmarks—most of its time is spent in kernel mode. This is due to ATS serializing threads to a central wait queue using pthread operations when contention is high. These operations quickly dominate runtime when ATS has to throttle execution for high contention applications. In other cases such as *Ssca2*, ATS is able to attain higher performance due to its extremely low scheduling overheads. This is in contrast to the BFGTS techniques which spend a fair amount of execution time in scheduling code for all benchmarks. Overall the overhead is less than that of PTS because the BFGTS technique appears to be making better predictions from the similarity metric, and the BFGTS-HW and BFGTS-HW/Backoff extensions further help to reduce execution overhead. This allows for better performance in the benchmarks *Kmeans*, *Ssca2*, *Vacation* and *Genome* which are overhead sensitive. In the case of BFGTS-HW/Backoff, it significantly reduces overhead of BFGTS-HW allowing for much higher performance on the *Genome* and *Vacation* benchmarks. In the benchmark *Intruder* scheduling is needed continuously, all methods spend a large amount of time scheduling to keep contention under control and provide forward progress.

5.3. Sensitivity studies

There are many parameters to explore that effect performance for the BFGTS technique. Presented here are the two parameters that had noticeable effects on performance: Bloom filter size, and Small transaction accounting interval.

5.3.1. Bloom filter size. Bloom filter size has a noticeable impact on performance for BFGTS-HW as shown in Figure 6(a). For benchmarks *Kmeans*, *Ssca2* and *Intruder*, the smallest Bloom filter size of 512bits performs best. These benchmarks are sensitive to the overhead imposed by doing the similarity calculations for large Bloom filters.

For *Delaunay*, *Genome*, and *Vacation*, these benchmarks prefer larger Bloom filters, on the order of 1024bits to 2048bits in size. They can tolerate the overhead of doing the similarity calculations when committing a transaction as it is offset by better predictions leading to better performance. As the Bloom filter sizes increase to greater than 4096bits, the overhead of doing the similarity calculations



(a) BFTGS-HW Bloom Filter Sensitivity

(b) BFGTS-HW/Backoff Bloom Filter Sensitivity

Figure 6. Speedups attained for different sized Bloom filters for BFGTS-HW and BFGTS-HW/Backoff for a 16 processor system.

dominates any performance increase had from better prediction accuracy.

The last benchmark, *Labyrinth*, appears insensitive to Bloom filter size until the largest Bloom filter (8192bits) is used. *Labyrinth* has very large transactions, and therefore large Bloom filters allow for better prediction accuracy and prevent overly pessimistic predictions. Because the transactions are large, it is able to amortize the cost of calculating similarity with 8192bit Bloom filters.

For the BFGTS-HW/Backoff contention manager, Figure 6(b) shows it is not as sensitive to Bloom filter size as BFGTS-HW. Because BFGTS-HW is turned on and off by BFGTS-HW/Backoff it is able to on average use larger Bloom filters. This is especially apparent for the *Vacation* benchmark where BFGTS-HW/Backoff is able to use 8192bit Bloom filters and approach the performance of BFGTS-No Overhead. On the other hand, BFGTS-HW is only able to amortize the cost of using a 2048bit filter as seen if Figure 6(a) and achieves significantly less performance due to lower prediction accuracy.

5.3.2. Small transaction calculation interval. As covered in Section 4.2.2, to eliminate some of the overhead of calculating similarity for transactions that are very small, a sweep of accounting intervals for similarity was performed. This consisted of testing the similarity calculations on every commit, every 10 commits, and every 20 commits. Updating on every commit affects overall average performance of BFGTS-HW, decreasing it to 20% better on average over PTS. Performing similarity calculations for small transactions every 10 gets an overall average performance of 23% better than PTS. As presented in the results here performing similarity calculations for small transactions every 20 gets an overall average performance of 25% better than PTS.

6. Conclusion

For transactional memory to be widely adopted, it has to be easy to work with. The study by Rossbach et al. [22] indicated that it is indeed easier to program with fewer concurrency errors compared to legacy techniques. Contention in TM is an important problem, particularly when novice programmers create large critical sections with irregular behavior. To achieve scalable performance contention must be minimized, but requiring the programmer to identify and fix these sources of contention by hand essentially eliminates the programmability benefits of TM. To remove the responsibility of discovering these sources of contention from the programmer, proactive transaction scheduling techniques have been proposed. They attempt to avoid contention before it happens; achieving better performance in high contention situations when reactive contention managers fail.

In this work we propose a new proactive scheduling technique called "Bloom Filter Guided Transaction Scheduling" (BFGTS). It uses Bloom filters to characterize transaction behavior to guide scheduling predictions coupled with simple hardware to accelerate high overhead operations. BFGTS-HW obtains a 25% speedup on average over PTS and up to 1.7x improvement on high contention benchmarks. Compared to ATS, BFGTS-HW obtains 35% better performance on average and up to 4.6x improvement on high contention benchmarks. A hybrid version, BFGTS-HW/Backoff can do better, getting 30% speedup over PTS and 40% over ATS on average.

References

[1] Intel 64 and IA-32 Architectures Software Developer's Manual. *Intel Developer Manuals*, 2, Nov 2008.

[2] M. Ansari, C. Kotselidis, K. Jarvis, M. Luján, C. Kirkham, and I. Watson. Advanced concurrency control for transactional memory using transaction commit rate. In *EUROPAR '08: Proc. 14th European Conference on Parallel Processing*, pages 719–728, Aug 2008. Springer-Verlag Lecture Notes in Computer Science volume 5168.

[3] M. Ansari, M. Luján, C. Kotselidis, K. Jarvis, C. Kirkham, and I. Watson. Steal-on-abort: Improving transactional memory performance through dynamic transaction reordering. In *HIPEAC '09: Proc. 4th International Conference on*

High Performance and Embedded Architectures and Compilers, pages 4–18, Jan 2009. Springer-Verlag Lecture Notes in Computer Science volume 5409.

[4] T. Bai, X. Shen, C. Zhang, W. N. Scherer III, C. Ding, and M. L. Scott. A key-based adaptive transactional memory executor. In *Proceedings of the NSF Next Generation Software Program Workshop*. Mar 2007. Invited paper. Also available as TR 909, Department of Computer Science, University of Rochester, Dec 2006.

[5] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt. The m5 simulator: Modeling networked systems. *IEEE Micro*, 26(4):52–60, 2006.

[6] G. Blake, R. G. Dreslinski, and T. Mudge. Proactive transaction scheduling for contention management. In *Micro-42: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 156–167, New York, NY, USA, 2009. ACM.

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

[8] J. Bobba, K. E. Moore, L. Yen, H. Volos, M. D. Hill, M. M. Swift, and D. A. Wood. Performance pathologies in hardware transactional memory. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*. Jun 2007.

[9] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multiprocessing. In *IISWC '08: Proceedings of The IEEE International Symposium on Workload Characterization*, Sep 2008.

[10] D. Christie, J.-W. Chung, S. Diestelhorst, M. Hohmuth, M. Pohlack, C. Fetzer, M. Nowack, T. Riegel, P. Felber, P. Marlier, and E. Rivière. Evaluation of AMD's advanced synchronization facility within a complete transactional memory stack. In *EuroSys '10: Proceedings of the 5th European conference on Computer systems*, pages 27–40, New York, NY, USA, 2010. ACM.

[11] C. Click. Azuls experiences with hardware transactional memory. In *In HP Labs - Bay Area Workshop on Transactional Memory*, 2009.

[12] S. Dolev, D. Hendler, and A. Suissa. Car-stm: scheduling-based collision avoidance and resolution for software transactional memory. In *Proceedings of the Twenty-Seventh Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 125–134. Aug 2008.

[13] A. Dragojević, R. Guerraoui, A. V. Singh, and V. Singh. Preventing versus curing: Avoiding conflicts in transactional memories. In *PODC '09: Proc. 28th ACM Symposium on Principles of Distributed Computing*, Aug 2009.

[14] A. Fog. Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs, 2010.

[15] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 289–300. May 1993.

[16] M. Kulkarni, L. P. Chew, and P. Keshav. Using transactions in delaunay mesh generation. *Workshop on Transactional Memory Workloads*, 2006.

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

[18] W. Maldonado, P. Marlier, P. Felber, A. Suissa, D. Hendler, A. Fedorova, J. L. Lawall, and G. Muller. Scheduling support for transactional memory contention management. In *PPoPP '10: Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 79–90, New York, NY, USA, 2010. ACM.

[19] L. Michael, W. Nejdl, O. Papapetrou, and W. Siberski. Improving distributed join efficiency with extended bloom filter operations. *Advanced Information Networking and Applications, International Conference on*, 0:187–194, 2007.

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

[21] H. E. Ramadan, C. J. Rossbach, O. S. Hofmann, and E. Witchel. Dependence-aware transactional memory. In *The 41st Annual International Symposium on Microarchitecture*. Nov 2008.

[22] C. Rossbach, O. Hofmann, and E. Witchel. Is transactional memory programming actually easier? In *WDDD '09: Proc. 8th Workshop on Duplicating, Deconstructing, and Debunking*, Jun 2009.

[23] D. Sanchez, L. Yen, M. D. Hill, and K. Sankaralingam. Implementing signatures for transactional memory. In *MICRO '07: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 123–133. IEEE Computer Society, 2007.

[24] W. N. Scherer III and M. L. Scott. Contention management in dynamic software transactional memory. In *Proceedings of the ACM PODC Workshop on Concurrency and Synchronization in Java Programs*, St. John's, NL, Canada, Jul 2004.

[25] W. N. Scherer III and M. L. Scott. Advanced contention management for dynamic software transactional memory. In *Proceedings of the 24th ACM Symposium on Principles of Distributed Computing*, Las Vegas, NV, Jul 2005.

[26] N. Sonmez, T. Harris, A. Cristal, O. S. Unsal, and M. Valero. Taking the heat off transactions: Dynamic selection of pessimistic concurrency control. In *IPDPS '09: Proc. 23rd International Parallel and Distributed Processing Symposium*, May 2009.

[27] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22th International Symposium on Computer Architecture*, pages 24–36, Santa Margherita Ligure, Italy, 1995.

[28] R. M. Yoo and H.-H. S. Lee. Adaptive transaction scheduling for transactional memory systems. In *SPAA '08: Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, pages 169–178, New York, NY, USA, 2008. ACM.

[29] C. Zilles and L. Baugh. Extending hardware transactional memory to support nonbusy waiting and nontransactional actions. In *Proceedings of the First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*. Jun 2006.