Proactive Transaction Scheduling for Contention Management

Geoffrey Blake
University of Michigan
Ann Arbor, MI
blakeg@umich.edu

Ronald G. Dreslinski
University of Michigan
Ann Arbor, MI
rdreslin@umich.edu

Trevor Mudge
University of Michigan
Ann Arbor, MI
tnm@umich.edu

ABSTRACT

Hardware Transactional Memory offers a promising high performance and easier to program alternative to lock-based synchronization for creating parallel programs. This is particularly important as hardware manufacturers continue to put more cores on die. But transactional memory still has one main drawback: contention. Contention is caused by multiple transactions trying to speculatively modify the same memory location concurrently causing one or more transactions to abort and retry its execution. Contention serializes the execution, meaning high contention leads to very poor parallel performance. As more cores are added, contention worsens. To date contention-manager designs have been primarily reactive in nature and limited to various forms of randomized backoff to effectively stall contending transactions when conflicts occur.

While backoff-based managers have been popular due to their simplicity, at higher core counts our analysis on the STAMP benchmark suite shows that backoff-based managers perform poorly. In particular, small groups of transactions create hot spots of contention that lead to this poor performance. We show these hot spots commonly consist of small sets of conflicts that occur in a predictable manner. To counter this challenge we introduce a dynamic contention management strategy that minimizes contention by using past history to identify when these hot spots will reoccur in the future and proactively schedule affected transactions around these hot spots. The strategy used predicts future contention and schedules to avoid it at runtime without the need for programmer input. Our experiments show that by using our proactive scheduling technique we outperform a backoff-based policy for a 16 processor system by an average of 85%.

Categories and Subject Descriptors

C.0 [**General**]: System architectures

MICRO'09, December 12–16, 2009, New York, NY, USA.

General Terms

Design, Experimentation, Performance

Keywords

Hardware transactional memory, Proactive scheduling, Software runtime

1. INTRODUCTION

Transactional Memory (TM) [15] promises to simplify concurrent programming by giving the programmer the abstraction of a critical section behaving as an atomic block of code. Instead of having to carefully compose the correct sequence and set of locks to protect a critical section, a programmer simply has to provide where to begin and end the critical section. TM also provides the performance advantages of fine-grained locking because a transaction effectively "locks" each word of memory touched during its execution.

These key advantages have encouraged many efforts to build prototypes and analyze the overall effectiveness of TM in hardware [13, 20, 31, 11, 24, 23, 6], software [28, 21, 14, 27], and hybrid implementations [19, 10]. This past research has laid down the blueprints for the necessary infrastructure to build a correct TM that executes critical sections as atomic units of work. This previous work has shown that transactions work just as well and in many cases better on parallel programs that were originally written with locks.

Even though initial research has shown TM to be very promising, new benchmarks have emerged to test TM systems more rigorously. In particular, benchmarks from the STAMP [9] suite have shown TM applications can suffer from severe performance degradation due to contention. Contention occurs when multiple transactions try to access or modify overlapping memory regions. When this happens, *contention management* decides how long to stall before restarting the transaction causing the conflict to ensure forward progress. Ineffective strategies for stalling and restarting can significantly degrade utilization of parallel resources.

Currently the most commonly used form of contention management is a variation on randomized exponential or linear backoff which determines how long a transaction is stalled before restarting after detecting a conflict. Randomized backoff is inadequate primarily because it reacts to conflicts and does not prevent them from recurring in the future under similar circumstances leading to worse than expected performance. To reduce contention some research efforts have considered strategies such as selectively marking cache lines which is used by software and hybrid TMs [10, 19],

open nesting with compensation routines as first proposed by McDonald et al. [18], and early release proposed by Skare et al. [29]. The reduction of contention achieved by these approaches requires careful annotation of parallel code sections by the programmer, undermining the ease-of-use advantage of TM over locking. To eliminate these added complexities, and maintain TM's main selling point as being easy to use, we propose our proactive scheduling contention manager. It is a dynamic and minimalistic contention manger that learns to recognize situations where conflicts are likely and acts to reduce contention in Hardware Transactional Memories (HTMs) before they occur by rescheduling transactions, thereby minimizing wasted computation. Unlike past methods to eliminate contention, this technique requires no programmer input or knowledge of its existence and is able to extract better performance on average over simpler contention managers.

Our technique features the following innovations:

- A minimal amount of additional hardware to enable logging of conflicting transaction pairs in a software runtime.
- A confidence based predictor written in software that assesses the probability of future conflict by analyzing the correlation of transaction pairs.
- An efficient user-space thread scheduler that operates independently of the operating system's thread scheduler.

Our proposed solution proactively schedules transactions, by identifying threads likely to conflict dynamically during runtime, and enforces a more appropriate schedule. In addition, our solution does not require code annotations by the programmer. In our experiments, we demonstrate the software proactive scheduler can increase performance by an average of 85% compared to a simple backoff based manager and reduce contention by 4-5x on average for 16 processor systems.

The rest of this paper is organized as follows. In Section 2 we illustrate our observations that transactions conflict in small predictable groups, leading to the hypothesis that transactions can be scheduled for better performance. Section 3 describes the implementation of our HTM system and the software proactive scheduling contention manager prototype. Section 4 analyzes our results. Section 5 discuses related work, and finally Sections 6, and 7 cover future work and present our conclusions.

2. MOTIVATION

Contention management design is an important consideration when building an HTM. The goal of an effective contention manager is to maximize the concurrency of the system by ensuring forward progress and preventing transactions from repeatedly aborting and restarting due to contention. Contention is not a large problem in most cases at small processor counts. But as systems scale to higher processor counts, the problem of contention between transactions is exacerbated. A common method to handle contention is randomized backoff. It is an extremely simple and low-cost contention manager. While it works well at low processor counts, we have found that as the number of processors and threads increase, the effectiveness of randomized backoff rapidly decreases. Referring to Table 1, we see the conflict rates for the STAMP benchmark suite running on an Eager Commit/Eager Conflict Detecting HTM, similar to LogTM [20] using randomized linear backoff for contention management. For all but the benchmark *Ssca2*, contention is a problem and limits scaling over sequential code. Of particular interest is the *Genome* and *Kmeans* benchmarks. For these two, contention becomes a large problem at 16 processors and in both cases the performance scaling reverses and is worse than the performance at 8 processors. Likewise for the *Intruder* benchmark, as at 8 processors its performance falls below that of the 4 processor configuration and at 16 processors this trend continues.

As shown in Table 1, the effectiveness of randomized backoff in managing contention degrades as the number of cores increases. We propose scheduling transactions in place of randomized backoff to manage contention. This idea leverages two key observations: most applications that benefit from parallel programming have large problem set sizes and are mostly throughput oriented, so there should always be another thread ready from the current program to do other independent work that can be swapped in to allow better forward progress. Another observation that we validate later is that transactions conflict mostly in small groups indicating there is extra parallelism to exploit by swapping threads. We use these observations to build our proactive scheduler to extract better performance dynamically at runtime. In Figure 1 we show a simple example of how scheduling can manage contention and improve performance. The example shows a two processor system trying to execute three transactions. In the backoff case, Tx2 tries to execute and conflicts with Tx1, the backoff contention manager will pick a backoff period to wait before retrying Tx2's execution. In this case the backoff period is too short, and Tx2 executes again but conflicts. The backoff policy increases the backoff time, Tx2 waits long enough for the conflict to clear and commits successfully. Tx3 then executes and commits. We propose that by using the past conflict history, a scheduler can predict that it is likely that Tx2 and Tx1 will conflict. The scheduler can suspend the thread trying to execute Tx2, and run Tx3 in its place. This improves performance by performing other useful work in place of a transaction that is likely to conflict with another concurrently executing transaction. Another way to think of this technique is that it is dynamically creating blocking synchronization (locks) or optimistic synchronization (transactions) when appropriate. Unlike backoff techniques that stall transactions to avoid conflicts, our technique can bring in new work to maximize useful computation instead of spending it stalling. We also maintain a history so blocking synchronization decisions do not immediately revert back to optimistic synchronization that would allow conflicts to reoccur.

2.1 Randomized Backoff Theory

To understand why randomized backoff contention managers may perform poorly, we investigated studies from the network domain that deal with similar problems of contention for a network link. In "*Performance Analysis of Exponential Backoff*" by Kwak et al. [16], the authors derive a mathematical model of an ethernet system to understand how it performs as the number of parallel transmitters contending for the ethernet wire is increased. As contention to transmit on the network increases, they found that the prob-

Table 1: Percent of all begun transactions that conflict and require restart and resultant speed-up over sequential execution for the STAMP benchmark suite using Randomized Linear Backoff for an Eager Commit/Eager Conflict Detecting HTM

Benchmark	2 CPU		4 CPU		8 CPU		16 CPU	
	Contention	Speedup	Contention	Speedup	Contention	Speedup	Contention	Speedup
Delaunay	28.3%	1.4	42.3%	1.9	53.8%	2.6	67.1%	3.0
Genome	0.3%	1.8	1.3%	2.4	4.7%	2.8	70.0%	**0.3**
Kmeans	0.1%	1.9	0.2%	3.9	4.5%	6.6	24.4%	**6.1**
Vacation	0.1%	1.9	0.3%	2.4	0.6%	2.9	1.8%	**2.9**
Intruder	2.5%	1.5	12.2%	1.5	36.4%	**0.9**	68.7%	**0.4**
Ssca2	0.0%	1.9	0.0%	3.6	0.0%	7.0	0.1%	12.7
Labyrinth	90.8%	1.2	96.8%	1.2	98.6%	1.1	99.5%	1.0

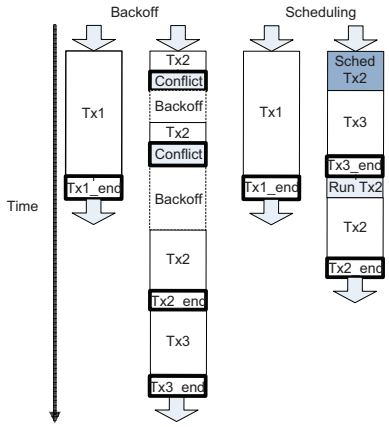

Figure 1: Example illustrating the advantages of proactively scheduling transactions to more effectively utilize parallel resources

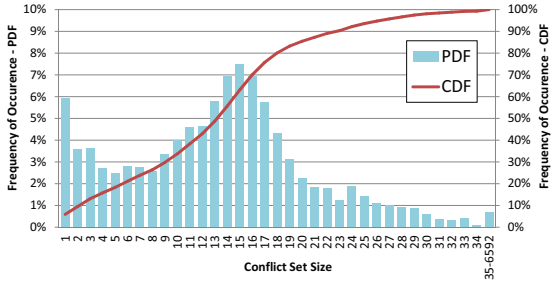

Figure 2: Histogram breakdown showing distribution of the number of individual unique conflicts each transaction sees during a program run of Delaunay

ability of successful transmission decreased to ∼35%. Qualitatively, this is what is seen with the STAMP benchmarks in our experiments. As the number of parallel transactions in the system increases, the probability of successfully completing becomes steadily smaller. The final conclusion of Kwak's et al. is that randomized backoff is the best solution for a system that has no knowledge of what is trying to run on a contended resource even though it performs poorly under high contention. However, in the case of transactions we do have knowledge, opening the possibility for a better solution. The next section will empirically show the existence of conflict patterns in benchmarks, which motivates the development of a predictor to manage contention and improve upon randomized backoff.

2.2 STAMP Benchmark Suite Profile

The STAMP benchmark suite is growing in popularity among researchers in the transactional memory field because of its long-term vision of how TM will likely be used in future applications. Recent works from Ramadan et al. [22], Bobba et al. [7] have turned to the STAMP suite to evaluate their systems. In early TM research, researchers used existing parallel programs from suites such as SPLASH2 [30], which were highly tuned parallel codes with very small transactions. But these are not very representative of future parallel programs because of this high optimization. In contrast, the STAMP suite uses large coarse grained transactions in relatively poorly tuned parallel code, which is considered more representative of future uses of TM, i.e. parallel programming for everybody else. This allows researchers to better predict future problems TMs may face. In particular, the STAMP benchmarks show rising amounts of contention as the number of processors is increased when using a randomized backoff technique.

Our scheduling technique depends on the number of unique conflicts experienced by each transaction to be small. A unique conflict is a conflict between two critical sections of separate threads. If the average number of unique conflicts is large per transaction, it would indicate that a new thread swapped in to replace a predicted conflict would also be likely to conflict, meaning the best case is to serialize. In this case, scheduling would be a high cost backoff algorithm because of the extra overhead scheduling requires to perform the scheduling decisions and therefore would be likely worse or equivalent to a simple backoff scheme. To test this property of small conflicting groups, we tagged each transaction in the program with a transaction ID (TxID) that was based off the program counter (PC) of the TM_BEGIN instruction, and the current thread ID. We then ran the STAMP benchmarks and recorded all conflicting pairs of TxIDs. In Figure 2 we show the histogram for unique conflicting transaction pairs for the Delaunay benchmark which has 6592 unique TxIDs. The histogram shows that the number of unique conflicts each transaction experiences is very small, on the order of tens. The small number of unique conflicts point to the existence of hot spots in the execution and with proper identification it is possible to schedule around these hot spots, thereby reducing contention and increasing performance. These types of hot spots are common across all the benchmarks tested.

3. SCHEDULER IMPLEMENTATION

The prototype contention manager is implemented as a user-space software runtime layer built on top of a Eager/Eager Transactional Memory system implemented in the M5 simulator [3] as shown in Figure 3. The HTM works in con-

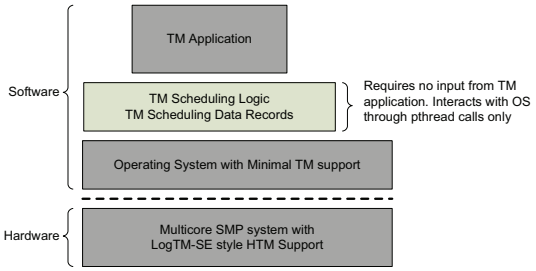

Figure 3: Hardware/Software stack of our proposed system

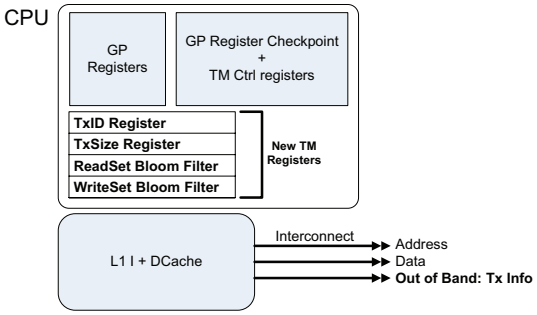

Figure 4: Additional registers and interconnect extensions to support proactive scheduling. New additions are bolded.

junction with the software runtime to implement our proactive scheduling contention manager. The scheduler is a *fully distributed* algorithm that each processor runs in parallel whenever a transaction wishes to begin execution. First, each processor looks at a snapshot of what transactions are currently executing on the system and gathers information about conflicts from the global transaction conflict graph stored in memory. Each processor then locally decides its probability of generating a conflict from information derived by these lookups and decides the appropriate course of action: swap in a new thread, stall briefly or begin execution. None of the processors running the scheduling algorithm explicitly communicate their intentions to each other, nor is the system snapshot necessarily consistent when viewed by multiple processors because it updated without using any form of synchronization. While enabling communication or globally consistent snapshots could be beneficial, the synchronization necessary to provide such facilities would be overly prohibitive. In the following subsections we will describe the pieces that constitute this system and how they fit together.

3.1 M5-TM

We build off an Eager/Eager transactional memory model developed inside the M5 full system simulator. The design closely resembles the original LogTM [20] and details of the base implementation can be found in [4]. To enable logging the necessary data for the software runtime we add additional functionality to the CPU, cache controllers and coherent interconnect.

The CPU and interconnect modifications are shown in Figure 4 and consist of additional registers and an out-of-band data channel which are bolded in Figure 4. The next set of registers are specific to our scheduling contention manger, though they could be used by other contention managers. The *TxID* register holds the ID of the currently running transaction. The *TxSize* register is updated on transaction commit with the total size of read and write sets. The final set of registers hold a read and write set summaries in the form of bloom filters [5] that can be accessed by the software scheduling runtime.

The second modification is to the coherent interconnect and the cache controllers. When the cache has to notify a remote transaction it must abort due to a conflict, the cache controller first gets the value of the *TxID* register from the CPU. Then it sends a response back to the remote processor containing this *TxID* over the interconnect as an out-of-band data response. When the remote cache receives this response, it passes the conflicting *TxID* value back to the remote processor to use to update the conflict information. This conflicting *TxID* is stored in one of the general purpose registers for direct use by the *abort routine* which is immediately vectored to by the CPU when a conflict occurs.

3.2 Proactive Scheduling Runtime

The software proactive scheduling runtime is implemented as a user-space thread scheduler. This design was chosen so the cost of calling into the scheduler at the start of every transaction is minimal. We use the pthread library provided in Linux to suspend threads and force the operating system scheduler to execute threads in an order determined by the proactive scheduler. The runtime uses three global data-structures and three main function routines to implement the distributed proactive scheduling algorithm.

3.2.1 Data Structures

There are three global data structures used by the scheduling runtime. These are the *CPU Status* Array, *Transaction Stats* table, and the *Conflicts* table. These data-structures provide the snapshot of the current transactions executing in the system and the conflict history in the form of a graph. An example of the required data structures for an eight processor system is shown in Figure 5.

CPU Status Array: The *CPU Status* array is a globally accessible array that is sized to the number of processors present in the system. As seen in Figure 5, it is an eight processor system so the array is of size eight. The array contains the information of what transaction is running currently on what processor by storing its TxID in that processor's corresponding entry.

Transaction Stats Table and Conflict Table: The next two data structures hold information about each individual transaction and also maintain information about past conflicts which is represented as a dependency graph using a full matrix representation. The *Transaction Stats* table is a global table shared by all threads, and each entry in the *Transaction Stats* table has its own *Conflict* table which is a row of the conflict graph matrix. Each *Conflict* table entry holds the saturating counter *conf* that represents the confidence of a conflict occurring in the future between a pair of transactions. A *Transaction Stats* table entry stores information such as the *AvgSize* variable, used to indicate the average historical runtime of this transaction represented using the overall number of memory addresses touched during execution. The table entry also holds a bloom filter representation of the most current successful commit of the transaction's read and write set. The purpose of these filters will be covered in the next section. The *Tx_waiting_on* vari-

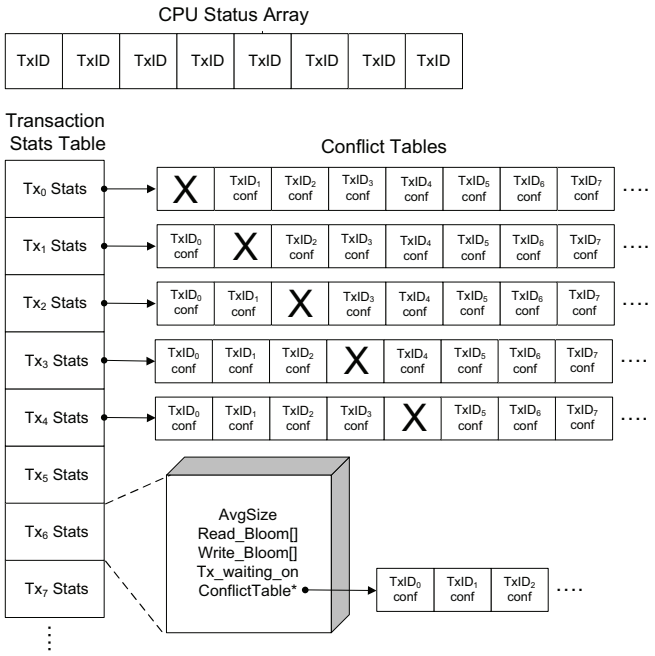

Figure 5: Data structure representation for an example 8 CPU system.

Example 1 Schedule Transaction Pseudo Code

```
1  void scheduleTx(int TxID)
2  {
3   start_schedule_loop:
4   for(int i=0;i<sizeof(cpuStatusArray);i++)
5   {
6     if(i!=ourCPU)
7     {
8       remoteTxID=cpuStatusArray[i];
9       if(confThreshold <
10          confProb(TxID,remoteTxID))
11      {
12        logTxWaitingOnVar(TxID,remoteTxID);
13        if(txSizeThreshold >=
14           checkSize(remoteTxID))
15        {
16          doSmallRandomBackoff();
17          break;
18        }
19        else
20        {
21          pthread_yield();
22          goto start_schedule_loop;
23        }
24      }
25    }
26  }
27  cpuStatusArray[ourCPU]=TxID;
28 }
```

able tracks which transactions this transaction has serialized behind. Its use will also be covered in more detail later.

In our design, the *CPU Status* array is implemented as a fixed sized array which is allocated at program start since the maximum number of processors is known. The *Transaction Stats* table is also allocated at program startup along with the *Conflict* tables because the number of unique transactions that can exist in the system is set at compile time and the maximum number of TxIDs is passed to the scheduling runtime library at program start. This requires a rather large memory footprint, on the order of $O(N^2)$, but offers an $O(1)$ time to access any part of the table or conflict graph matrix, reducing the overhead for each invocation of the scheduler. In our experiments our tables grew to a maximum of 50MB for benchmarks with a large number of TxIDs, and could only be implemented in software. However, more space efficient representations can be constructed.

3.2.2 Scheduler Algorithm Implementation

The scheduler has three main routines that form the main portion of the distributed algorithm. These functions work to schedule transactions, update the conflict graph, update the current snapshot of executing transactions, and update transaction statistics. The main functions are `scheduleTx()`, `txConflict()` and `commitTx()` and are described below. The scheduler is a parallel program in its own right, any of these three routines can be executed by any or all of the processors concurrently.

scheduleTx(): The `scheduleTx()` function is called before the start of any transaction and is shown in Example 1. It is rather simple and works by scanning the *CPU Status* array for TxIDs that could potentially conflict with the TxID wanting to execute. We decide if a conflict exists by indexing into the conflict graph matrix, using `TxID` as the row index and the `remoteTxID` as the column index to get a confidence value. If the confidence value is below the `confThreshold` (in our experiments the `confThreshold` is set to 5), the algorithm continues scanning, otherwise it decides how to serialize the transaction. If a conflict is predicted the function then decides if the conflicting transaction is "small" or "large" by indexing the *Transaction Stats* table. If the transaction is large, which is done by looking at the *AvgSize* then the scheduling function calls `pthread_yield()` to force the currently running thread to the back of its run queue in the Operating System (OS). The OS will then swap in a new thread that will try to execute its transactions. Upon return from `pthread_yield()`, the function restarts the scheduling process. If the transaction is decided to be small then a simple random backoff is initiated to stall the current transaction for a short while before letting it execute. This is done because calling `pthread_yield()` is expensive and unnecessary for short transactions. When the transaction finds it is not predicted to conflict with any other running transaction in the system, it sets its TxID in the *CPU Status* array and executes.

It is important to note that all the processors could begin running this routine at the same time because it is a *distributed* algorithm. Because there is no explicit synchronization or communication among processors i.e. locks or message passing, the scanning of the *CPU Status* array may yield stale information as the processors running the routines are in a benign data race to complete their scans and update their respective entries in the *CPU Status* array. This may lead to unintended conflicts because each processor may schedule conflicting transactions without realizing it due to inconsistent views of the *CPU Status* array caused by these data races. Still, this is desirable over inserting synchronization to only allow one writer at a time to the *CPU Status* array because the cost of such synchronization is high

Example 2 Conflict Handling Pseudo Code

```
1 void txConflict(int TxID, int confTxID)
2 {
3   cpu_status_array[ourCPU]=NO_TX;
4   incConflictProb(TxID, confTxID);
5   incConflictProb(confTxID, TxID);
6   if (txSizeThreshold >=
7       checkSize(confTxID))
8     doSmallRandomBackoff();
9 }
```

Example 3 Commit Transaction Pseudo Code

```
1 void commitTx(int TxID)
2 {
3   updateBloom(TxID);
4   updateAvgSize(TxID);
5   cpu_status_array[ourCPU]=NO_TX;
6   int TxWaitingOn=checkWasSerialized(TxID);
7   if (TxWaitingOn!=NO_TX)
8   {
9     if (intersectBlooms(TxID, TxWaitingOn))
10      incConflictProb(TxID, TxWaitingOn);
11    else
12      decConflictProb(TxID, TxWaitingOn);
13  }
14 }
```

and also allows us to not worry about deadlock because we do not use synchronization. On the other hand, starvation could happen, but throughout our experimentation we experienced no issues with starvation. We leave it as future work to investigate the implications starvation may have on our technique.

txConflict(): When a transaction conflicts with another transaction, the transaction is first rolled back. Then the `txConflict()` routine as shown in Example 2 is called to update the conflict graph matrix of the transaction pair that conflicted. The routine accesses each transaction's *Conflict* table and looks in the entries for the respective TxIDs. The conflicting TxID is obtained as discussed in Section 3.1, the processor stores the ID in a general purpose register that can be accessed by our routines easily. If the conflict has never been seen before, the confidence is initialized to a default value (in our case we set the default value to 5, and our counter saturated at 10), otherwise the confidence counter is incremented.

commitTx(): On commit, transactions call the `commitTx()` function which is shown in Example 3. In this function statistics such as average size and the current bloom filter are updated. First the thread erases its entry in the *CPU Status* array. Next the thread saves its current bloom filters. Finally the committed transaction checks to see if it had been waiting on another transaction by checking the *TxWaitingOn* variable in its *Transaction Stats* table entry. If it was serialized behind another transaction, it obtains the most current bloom filters from the table entry pointed to by *TxWaitingOn* to compare against its own. If the two summaries intersect, the thread increments the confidence of conflict with that transaction, otherwise it decrements the confidence. This last part using bloom filters to update confidence is vitally important as it is the method by which we identify transactions that have diverged and can predict that they will no longer conflict, thereby allowing pairs of transactions to resume using optimistic synchronization instead of pessimistic synchronization.

Table 2: M5 Full System Simulator Parameters

Feature	Description
Processors	1-16 cores Alpha ISA, 1 IPC at 2GHz
L1 Caches	Private Data and Instruction Caches, 64kB, 2-way associative, 64-byte line size, 1 cycle latency
Interconnect	Shared bus at 2GHz
L2 Cache	Shared 64MB, 16-way associative, 64-byte line size, 10 cycles latency
Main Memory	2048MB, 100 cycles latency
Linux Kernel	Modified v2.6.18

3.3 Hybrid Proactive Scheduling

We also developed a lightweight *hybrid* scheme to account for low contention cases where the proactive scheduling algorithm was too expensive. This is shown in Section 4 with the *Ssca2* benchmark. The hybrid scheme starts using backoff as its contention manager. During execution it keeps track of the global conflict rate. If the conflict rate reaches a preset threshold (5% in our tests), it will switch to using proactive scheduling instead of backoff.

4. RESULTS

In this section we present our results for the proactive scheduling runtime. We find that proactively scheduling transactions using our technique outperforms the simpler randomized backoff contention management scheme by an average of 85% and reduces contention by 4-5x on average for our largest system using 16 processors. This strongly indicates that contention managers can do much better than current ad hoc methods. We accomplish this by trading some scheduling overhead for a reduction in re-execution of transactions and elimination of backoff time. In the one corner case, where contention is near zero, the overhead of our scheduling technique leads to lower performance than backoff. We show that a simple adaptive technique can gain back most of the performance lost, without appreciably affecting the cases where scheduling is preferred. We will also show sensitivity and predictor accuracy studies.

4.1 Experimental Setup

To test the scheduling runtime, we model the multicore architecture presented in Table 2. The base TM system provides perfect conflict detection, but does keep track of a realistically sized bloom filter to be used by the scheduling software. The baseline contention manager is a randomized linear backoff algorithm that uses a Polite conflict resolution policy [25]. The backoff algorithm does not unboundedly grow the backoff period, it linearly grows to a maximum backoff period. On commit the current backoff period is reset to a small initial value. Our testing methodology runs ten different executions of each benchmark with small random variations added to the memory latencies to get a representative average of performance.

4.2 Benchmark Setup and Parameters

We test seven benchmarks from the STAMP [10, 9] benchmark suite. We modified these benchmarks to support unique transaction IDs to use in scheduling, as well as the proper infrastructure to support our TM simulator. Table 3 summa-

Table 3: STAMP Benchmark descriptions and input parameters.

Benchmark	Description and Parameters
Delaunay	Refines a 2D mesh of triangles using Delaunay refinement. Input "-i inputs/large.2 -m30 -t64"
Genome	Genome sequencing benchmark. Input "-g4096 -s32 -n524288 -t64"
Kmeans	Kmeans clustering algorithms. Input "-m20 -n20 -t0.05 -i inputs/random50000_12 -p64"
Vacation	Simulates a multi-user database, modeled as a Red-Black tree. Input "-n8 -q10 -u80 -r65536 -t131072 -c64"
Intruder	Signature-based network intrusion detection benchmark that captures and reassembles packet streams for scanning. Input "-a10 -l32 -n8192 -s1 -t64"
Ssca2	An efficient graph construction algorithm using adjacency arrays and auxiliary arrays. Input "-s15 -i1.0 -u1.0 -l3 -p3 -t64"
Labyrinth	A transactional version Lee's routing algorithm[17] through a maze. input "-i inputs/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 4: Summary of speedups for 16 processors for Randomized Backoff and Proactive Scheduling.

Benchmark	Backoff Speedup	Proactive Scheduling Speedup	
Delaunay	3.0	4.1	+36%
Genome	0.4	3.7	+825%
Kmeans	6.1	9.8	+61%
Vacation	2.9	4.0	+38%
Intruder	0.4	1.2	+210%
Ssca2	12.5	9.3	–26%*
Labyrinth	1.0	1.2	+17%
GeoMean			**+85%**

*Corner case, contention <0.1% (Hybrid predictor has speedup of 10.3)

rizes the benchmarks and the input parameters used. Note that all the benchmarks over-commit the system with more threads than processors (64 threads, maximum of 16 processors) to enable different threads to be swapped in by our scheduler when a conflict is predicted thereby keeping all processors busy. We also overcommit the baseline system to allow a fair comparison. It is reasonable to assume that future programs will have more threads than CPUs to expose as much parallelism as possible. The input parameters selected strike an even balance between large input set size and reasonable simulation time.

4.3 Speedup

For our results we show three data points for discussion: the baseline system with randomized linear backoff, our software based proactive scheduling runtime, and the simple hybrid scheme that can use backoff in low contention cases or switch to our proactive scheduler if contention becomes high. The speedups and percentage of contention for the STAMP benchmarks are presented in Figure 6. Table 4 shows the absolute percent improvement (or loss) for our software scheduler over backoff for sixteen processors. We omit the hybrid numbers for brevity but as seen in Figure 6 it is similar to the proactive scheduler in all but the *Ssca2* benchmark. The trends seen for backoff match those of published results for similar eager/eager TMs using the STAMP suite.

The main trend to note in the speedup results of Figure 6 is that in all but one corner case, *Ssca2*, proactive scheduling outperforms backoff at least by sixteen processors. In particular, for the *Delaunay* benchmark proactive scheduling is always better than backoff due to the large amount of contention present. For *Genome*, *Kmeans*, *Vacation*, and *Intruder* Figure 6 shows that backoff plateaus or reverses at high processor counts. On the other hand, for those same benchmarks, proactive scheduling shows a trend towards increased scaling that should continue for even higher processor counts given the drastic reduction in conflict rates seen by the bars in Figure 6. For *Labyrinth* the high contention implies that neither contention manager can do better than serialize all transactions. The proactive scheduler does marginally better because it is able to prevent wasted work over the backoff type manager by identifying that almost all the transactions need to be stalled (though the proactive scheduler does find a little parallelism in this benchmark). The backoff manager, on the other hand, keeps restarting transactions. In this situation one could foresee using this dynamic identification of completely serialized operations to save power by turning off cores for example.

For, *Ssca2*, backoff beats proactive scheduling in all processor configurations. This is a corner cases due to the extremely low contention present. For *Ssca2*, the low contention rate requires little to no contention management and any overhead degrades performance scaling. In the scheduling case for *Ssca2*, it is always incurring extra overhead from running the `scheduleTx()` routine at the beginning of every transaction even though the contention is extremely low. This accounts for the degraded performance seen for the scheduling case in Figure 6(f). With an adaptive policy, we can gain back most of the performance lost in the *Ssca2* benchmark as seen in Figure 6(f). Even with this corner cases, the software scheduler achieves a 85% average performance gain over backoff as shown in Table 4.

The adaptive hybrid policy also helps with the benchmark *Kmeans* for lower processor counts. It gains back all the performance lost at 2,4, and 8 processors and is comparable to proactive scheduling at 16 as seen in Figure 6(c). This shows that an adaptive method is helpful when contention is low at more modest core counts. In all the benchmarks studied, the hybrid method is comparable to proactive scheduling, usually performing slightly worse due to lost transaction history information during the period backoff is used before switching to proactive scheduling.

4.4 Contention

Referring back to Figure 6 we see a strong correlation between increase in performance and reduction in contention. For all but *Ssca2*, a reduction in contention leads to better performance. The most marked examples of this are *Kmeans* and *Genome*, where at sixteen processors the large amount of contention inhibits scaling resulting in worse performance than lower processor counts. Scheduling the transactions proves beneficial as the decrease in contention, and hence less re-execution of transactions allows continued scaling.

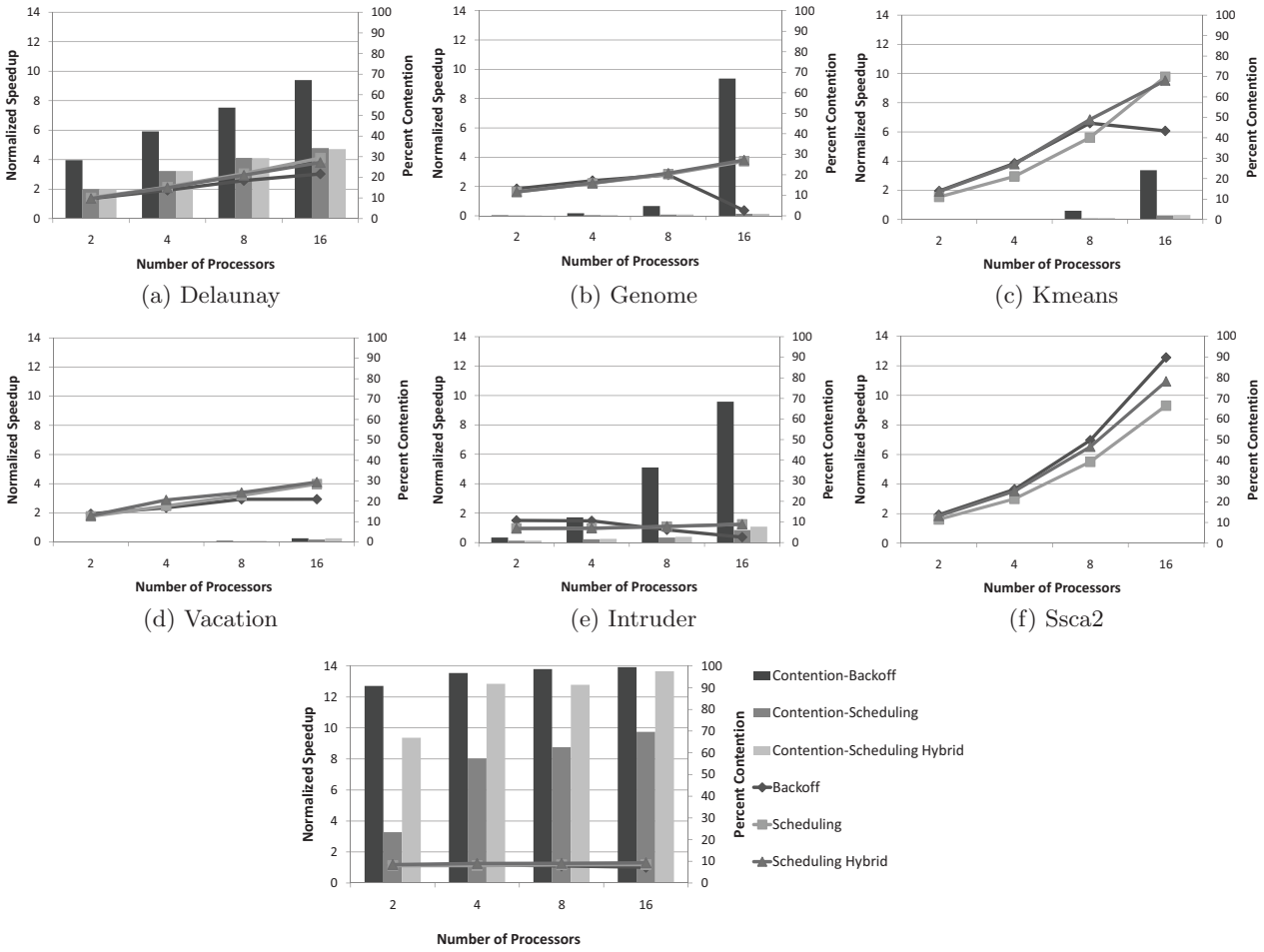

Figure 6: Normalized speedup over sequential execution and percent contention for the STAMP benchmarks. Speedup is represented by the lines, and the values are on the left y-axis, higher is better. Contention is represented by the bars and values are on the right y-axis, lower is better

4.5 Time Breakdown

A runtime breakdown of the STAMP benchmarks at sixteen processors is shown in Figure 7 for proactive scheduling and backoff. The hybrid scheme is omitted for brevity as the results are very similar to the standard proactive scheduling algorithm. The time is broken down as non-transactional, Operating System (OS), transactional, and scheduling/backoff. The goal of our scheduling work is reduce the transactional portion of backoff, representing a reduction in re-executed code. At the same time we are trading off backoff time for scheduling time. In most cases scheduling shows a reduction in both transactional and scheduling/backoff time due to decreased amounts of contention.

In the *Delaunay*, *Genome*, *Kmeans* and *Intruder* benchmarks, it is clear from the figure that scheduling is reducing the time spent in the re-execution of transactions and the usage of the contention manager. For the corner case we can see why it performs poorly. In *Ssca2*, we see that the software scheduler is spending time in the scheduling routines, adding overhead to each transaction. This extra overhead is in turn leading to the worse performance.

For the *Vacation* benchmark, it is hard to see that the amount of time spent restarting transactions is slightly less than the scheduling software case which is also indicated by Figure 6(d) where the amount of contention is less for the scheduling software, leading to better performance.

4.6 Sensitivity Analysis

Five sensitivity studies were performed by varying parameters that can affect the scheduling algorithm. These parameters were: prediction latency, conflict threshold, bloom filter size, L2 cache size and L2 latency. In this section we will present the parameter that affected execution the most, prediction latency. The studies for conflict threshold and bloom filter size showed little to no sensitivity so discussion and data is omitted for brevity. We will briefly discuss cache size and latency sensitivity at the end of this section.

In Figure 8 we analyze the sensitivity of the benchmarks to the latency of the scheduler. The latency is varied from 1 to 1,000,000 cycles. The purpose of this study was to understand what impact scheduler execution time has on overall performance of the benchmarks as a whole. This is useful in understanding how future hardware assist mechanisms may be able to improve performance further. This study was accomplished by implementing the proactive scheduler in the M5 simulator. For general reference, the average ob-

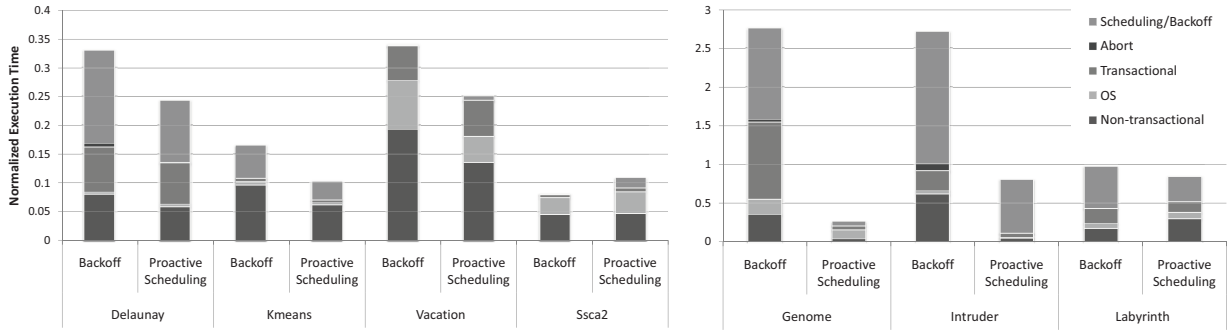

Figure 7: Time breakdown for the STAMP benchmarks running on 16 processors. Y-axis shows the normalized time relative to sequential execution, lower is better.

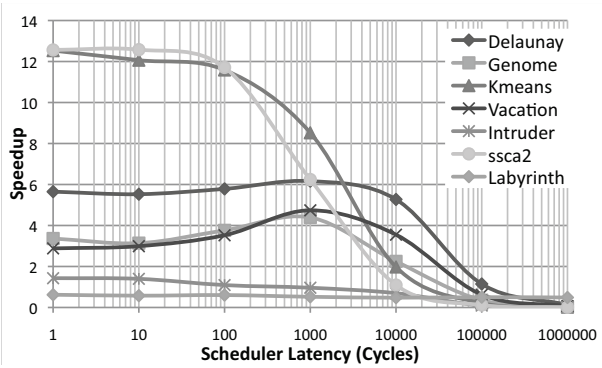

Figure 8: Sensitivity to predictor latency for 16 processor system. Speedup relative to sequential execution is presented.

served latency for the software implementation results in Section 4.3 of the proactive scheduler was anywhere from 500 to 10,000 cycles depending on the benchmark.

Ssca2, the low contention corner case, is highly sensitive to scheduler latency. This happens because there are no conflicts to amortize the cost of the scheduler overhead. On the other hand, *Labyrinth*, the high contention corner case, is not sensitive to scheduler latency. For some benchmarks like *Delaunay*, *Genome* and *Vacation* the graph is non-monotonic. As seen in Figure 8 the highest speedup is achieved around latencies of 1,000 cycles. Longer latencies than 1,000 cycles increase runtime due to the latency dominating, whereas shorter latencies also increase runtime due to temporal characteristics of the transactions' executions. For example, in the case of *Vacation*, the better speedup at higher latency is due to the properties of its red-black tree operations being done in transactions. Updating the tree spends the majority of time walking the tree, and doing writes at the end. In most cases, these operations can proceed in parallel if the transaction executions are staggered in time so early transaction commit before the later transactions read the same section. In other words it is seeing a type of "pipelining" effect. This staggering is being provided by the longer latency by letting the proactive scheduler predict more optimistically because the *CPU Status* array is slightly stale due to similar race conditions like those present in the software scheduler. This leads us to the conclusion that running transactions in parallel also depends to a first degree on the timing of their executions. The *Genome* and *Delaunay* benchmarks have similar properties to *Vacation* and therefore similar reasons as to why they prefer longer latencies for scheduling. Currently we only determine how to schedule a transaction with regard to their spatial characteristics, i.e. the memory locations they access. Temporal aspects will need to be considered as well to generate better schedules.

Table 5: Proactive Scheduler Misprediction rates for 16 processors.

Benchmark	%Mispredict Total	%Mispredict Parallel	%Mispredict Serial
Delaunay	21.1%	18.4%	2.7%
Genome	11.1%	2.6%	8.5%
Kmeans	7.6%	2.3%	5.3%
Vacation	2.1%	2.1%	0.0%
Intruder	16.8%	5.3%	11.5%
Ssca2	0.3%	0.3%	0.0%
Labyrinth	17.9%	17.9%	0.0%

Level 2 cache size and latency showed little sensitivity when analyzed. Still it is an important parameter to discuss because the *proactive software scheduler* depends on the ability to access its rather large prediction data quickly. For cache size we saw little sensitivity. For both the scheduling and backoff based managers, the slow down followed the same trend as cache size decreased: backoff lost performance at the same rate and always performed worse for each benchmark except *Ssca2*. For cache latency, sensitivity was also low. Scheduling and backoff lost performance again at the same rate as the L2 got slower (we tested latencies from 10-100 cycles). Scheduling again beat backoff in all cases except for *Ssca2*. We omitted the data as both cache parameters exhibited little sensitivity.

4.7 Predictor Analysis

Misprediction rates for the proactive scheduler are shown in Table 5. A mispredict is when the scheduler either decides to allow the transaction to run in parallel and a conflict occurs, or when the scheduler decides to serialize two or more transactions when they could have been run in parallel. These mispredicts lead to additional runtime, so the lower the mispredict the rate, the closer we are to an optimal scheduling. The mispredict serial information was gathered using a bloom filter that was 2048-bits in size, which for the average transaction size observed in the benchmarks has a 1.6% degree of accuracy [5].

For the benchmarks *Vacation*, *Ssca2*, and *Labyrinth*, all of the mispredicts were attributed to the scheduler being overly optimistic i.e, predicting parallelism that does not

exist, but never predicting conflicts that do not exist. For the remaining benchmarks there is a mix of both forms of misprediction. For example, the *Kmeans* benchmark has a total mispredict rate of 7.6% with roughly two thirds of the predictions mispredicting a future conflict between the transactions. This indicates the scheduling algorithm as it stands currently, is sometimes slow to learn when transaction behavior has diverged. We believe this is due to the low sampling rate of the bloom filters to update conflict confidences. The scheduler only inspects them if a serialization among transactions was predicted and for the most current transaction that was serialized against. As shown in Section 4.6 the scheduler is highly sensitive to algorithm latency, meaning more complicated prediction schemes that increase scheduler overhead would adversely affect performance. We believe that adding additional hardware to our design in the future may help reduce this type of overly pessimistic prediction, while minimally impacting scheduler runtime.

5. RELATED WORK

Contention managers have primarily been studied in Software Transactional Memories (STM) by Scherer et al. [25, 26]. The managers presented in these works have been reactive in nature, fixing conflicts as they happen. The process for deciding how much to backoff and which transaction to abort on a conflict is decided by a set of heuristics. Work has attempted to intelligently schedule transactions to avoid conflicts. Work by Bai et al. [2] proposed such a technique for software TM. The technique did work well, but the main disadvantage was the need for the transaction manager to have advance knowledge of the benchmark and its transactions. This meant the manager had to be changed whenever the benchmark changed. Another STM that is close to this idea is by Aydonat et al. [1]. They use the idea of serialization graph testing to perform contention management up front. Another group has also used the idea of scheduling in an STM, called CAR-STM [12], but they too require the use of programmer provided hints. A paper by Zilles et al. [33] is also similar to ours, in that they propose suspending conflicting transactions to avoid busy waiting. This work does not employ hysteresis as this stalling only occurs when two transactions conflict and does not work to prevent future conflicts.

In hardware transactional memory, contention management has so far been mainly overlooked. The primary form of contention management to date has been some variant of exponential backoff. There was a proposal for a smarter contention manager by Bobba et al. [8], but the proposals were not investigated in depth as this was not the focus of the paper. Rather the paper pointed to the types of pathologies present in HTMs, and did in depth analysis of what these pathologies are and why they are important. In the work by Hammond et al. [13], scheduling is used to order transaction commits when doing speculative loop parallelization. The scheduling was limited to just either run the transactions in loop order, or completely out of order. In work by Rossback et al. [24] scheduling is done in the OS and is with regard to thread priority as assigned by the OS. Work by Yoo et al. [32] is similar in that they develop a learning contention manager to schedule threads. They use a centralized technique to hold threads from executing in a queue when they measure a global contention rate that is higher than some threshold to enable currently executing transactions to complete. Their method is evaluated as a central hardware queue for a HTM and as a central software queue for a STM. They differ primarily in that their system runs less than the number of threads that can be supported in the system concurrently—not finding other independent work—and the algorithm is centralized rather than distributed like ours. In a recent work by Ramadan et al. [22], they look at contention management from a different angle. Instead of designing a new contention manager like in this work, they add to the cache coherence policy to enable transactions to share speculative data. This allows the authors to eliminate conflicts that would otherwise occur. This work has the same goal as ours, increasing concurrency, but takes a different approach.

6. FUTURE WORK

The next step is to develop a fully distributed hardware implementation of the proactive scheduler. While we roughly studied the effects of having hardware acceleration by modeling a zero-latency version of our software scheduler in M5 for our latency sensitivity study, it is not representative of real hardware. The current software scheduler has rather high memory requirements, far higher than what can be put into hardware. The current model also does not take advantage of the opportunities afforded by hardware, like a higher sampling rate of transaction characteristics which cannot be done in software currently due to the overheads involved.

The scheduling algorithm presented here also requires some method to encode and evaluate the temporal characteristics of a transaction. As was seen in the latency sensitivity studies, three benchmarks in particular were sensitive to timing and performance could be improved by actually increasing latency. This implies additional performance opportunities by only partially serializing transactions to increase throughput.

Other future work would include investigating other ways to use the conflict information gathered during a program run. It may be useful for parallelizing compilers and even for debugging to help pinpoint bottlenecks that are serializing execution more than anticipated. It could also conceivably be used for power reduction as cores could be shut down if sufficient parallelism can not be discovered during runtime.

7. CONCLUSION

We have shown that the contention manager makes a large impact on system performance. Simple contention managers such as randomized backoff are not adequate for managing high amounts of contention in a transactional memory system with a large number of processors. A well designed and intelligent contention manager is more effective at handling large amounts of contention. We have shown that by using past conflict history to form predictions on future conflicts, the system better utilizes the parallel resources in a multicore system. More importantly this technique requires no additional programmer input, thereby preserving the ease-of-use selling point transactional memory needs to be widely adopted.

In this paper we have developed a novel software based transaction scheduler. It effectively manages the concurrent resources in the system by scheduling transactions with low probability of conflicting to run concurrently. Even with

it's moderate runtime overhead, the software prototype performs much better than randomized backoff as the number of processors is increased. On average we can increase performance by 85% and reduce contention by 4-5x on average for 16 processors. We believe this work opens up new areas of research in TM for building dynamic systems that provide even greater performance.

8. ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers for their suggestions and comments. This work was supported by ARM Ltd.

9. REFERENCES

[1] U. Aydonat and T. Abdelrahman. Serializability of transactions in software transactional memory. In *Proceedings of the 3rd ACM SIGPLAN Workshop on Transactional Computing*. Feb 2008.

[2] 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, December 2006.

[3] 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.

[4] G. Blake and T. Mudge. Duplicating and verifying logtm with os support in the m5 simulator. *Workshop on Duplicating, Deconstructing and Debunking*, 2007.

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

[6] C. Blundell, E. C. Lewis, and M. M. K. Martin. Unrestricted transactional memory: Supporting i/o and system calls within transactions. Technical Report CIS-06-09, Department of Computer and Information Science, University of Pennsylvania, Apr 2006.

[7] J. Bobba, N. Goyal, M. D. Hill, M. M. Swift, and D. A. Wood. Tokentm: Efficient execution of large transactions with hardware transactional memory. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*. Jun 2008.

[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 multi-processing. In *IISWC '08: Proceedings of The IEEE International Symposium on Workload Characterization*, September 2008.

[10] C. Cao 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 *Proceedings of the 34th Annual International Symposium on Computer Architecture*. Jun 2007.

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

[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. August 2008.

[13] L. Hammond, B. D. Carlstrom, V. Wong, M. Chen, C. Kozyrakis, and K. Olukotun. Transactional coherence and consistency: Simplifying parallel hardware and software. *IEEE Micro*, 24(6), Nov-Dec 2004.

[14] M. Herlihy, V. Luchangco, M. Moir, and I. William N. Scherer. Software transactional memory for dynamic-sized data structures. pages 92–101, Jul 2003.

[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] B.-J. Kwak, N.-O. Song, and L. Miller. Performance analysis of exponential backoff. *Networking, IEEE/ACM Transactions on*, 13(2):343–355, April 2005.

[17] C. Lee. An algorithm for path connections and its applications. In *IRE Transactions on Electronic Computers*, 1961.

[18] A. McDonald, J. Chung, D. C. Brian, C. Cao Minh, H. Chafi, C. Kozyrakis, and K. Olukotun. Architectural semantics for practical transactional memory. pages 53–65. Jun 2006.

[19] M. Moir. Hybrid transactional memory, Jul 2005. Unpublished manuscript.

[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] M. Olszewski, J. Cutler, and J. G. Steffan. Judostm: A dynamic binary-rewriting approach to software transactional memory. In *Parallel Architecture and Compilation Techniques, 2007. PACT 2007. 16th International Conference on*, pages 365–375. IEEE, 2007.

[22] 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.

[23] H. E. Ramadan, C. J. Rossbach, D. E. Porter, O. S. Hofmann, A. Bhandari, and E. Witchel. Metatm/txlinux: transactional memory for an operating system. *SIGARCH Comput. Archit. News*, 35(2):92–103, 2007.

[24] C. J. Rossbach, O. S. Hofmann, D. E. Porter, H. E. Ramadan, B. Aditya, and E. Witchel. Txlinux: using and managing hardware transactional memory in an operating system. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 87–102, New York, NY, USA, 2007. ACM.

[25] 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.

[26] 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.

[27] N. Shavit and D. Touitou. Software transactional memory. In *Proceedings of the 14th ACM Symposium on Principles of Distributed Computing*, pages 204–213. Aug 1995.

[28] A. Shriraman, M. F. Spear, H. Hossain, V. Marathe, S. Dwarkadas, and M. L. Scott. An integrated hardware-software approach to flexible transactional memory. In *Proceedings of the 34rd Annual International Symposium on Computer Architecture*. Jun 2007.

[29] T. Skare and C. Kozyrakis. Early release: Friend or foe? In *Workshop on Transactional Memory Workloads*. Jun 2006.

[30] 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.

[31] L. Yen, J. Bobba, M. M. 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 *Proceedings of the 13th International Symposium on High-Performance Computer Architecture(HPCA)*. Feb 2007.

[32] 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.

[33] 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.