ABSTRACT

Shared data structure operation is one of the fundamental performance bottleneck in multithreaded processors. The simplest way to keep the correctness of the critical sections is mutual lock. By serializing the critical sections of threads, the critical section is guaranteed to be manipulated by single thread at any time. Dynamically, unnecessary serialization greatly hurt performance. Transactional memory that appear to execute atomically and isolation but may actually execute in parallel if there is false sharing. When the critical section do not require mutual exclusion, transactional memory removes unnecessary lock-induced serialization and enable multiple concurrent threads execution. While enjoying the concurrent benefit, the transactional memory must check conflict across all concurrent threads. A conflict indicates that the critical section is not lock-free and should avoid executing in parallel. In transactional memory, fallback or abort are common solutions for conflict, which are essentially recovery back to the point before critical sections. Due to fact that both write data and cpu internal states are required to recover back, the recovery usually takes a great amount of computing resources and computing time. In this paper, a hardware transactional memory predictor is introduced. The predictor dynamically predicts the target for critical section, that is lock and unlock, or transactional memory. The predictor exploits parallel execution of critical sections as transactional memory and tries to avoid conflicts and recovery. The predictor shows that multi thread processors have higher performance than lock and transactional memory.

keywords: Transactional Memory, hardware lock elision predictor, lock and unlock, multithreaded program

1 INTRODUCTION

Today’s processor has already enter the multicore multithread era with proliferation from data center processors to embedded mobile processors. Modern processor explicitly developed to support multithreaded
software in hardware. With such high performance multi thread support, software developers are expected to develop programs that could exploit these multithreaded hardware.

Synchronization mechanisms in multithreaded programs, such as lock and unlock, are required to guarantee threads have exclusive access to shared data structure, which are the critical section between the lock and unlock. A thread executes the critical section after acquiring its lock and before releasing its lock. All other threads must wait for lock until the first thread has completed its critical section and release its lock. Execution of critical sections in different threads are forced serialized and thus making the all critical sections appear to execute atomically. The drawbacks of lock and unlock is that concurrent access to a shared data structure by multiple threads in a critical section might not conflict. In many cases, different threads can operate on different parts of shared data structure, but does not because lock and unlock does not exploit parallel execution.

Transactional memory can be used to remove the unnecessary locks. The transactional memory make the lock-free synchronization as easy to use as conventional mutual exclusion lock. The transactional memory exploits multiple threads execution by always try to execute in parallel. To identify necessary lock, transactional memory must check data conflict in critical section. Once a conflict is detected, the thread must undo data write in the cache and recover CPU’s internal general registers files and internal states to the point before critical section. The conflict thread must recover as the thread has not executed. The recovery process takes a great amount a CPU computation time and resources. Therefore, for lock critical sections where transactional memory inevitably have to abort all the time and obtains worse performance than lock and unlock.

Transactional memory predictor proposed in this paper tries to do transactional memory and avoid aborts as often as possible. The transactional memory predictor predicts target based on previous history. With our transactional memory predictor, the processor provides higher performance than both lock and transactional memory and provides simple interface to the programmer.

The section 2 presents background of this paper, we introduce lock and lock, and transactional memory. Section 3 detail introduces our transactional memory predictor. Section 4 presents hardware predictor implementation and evaluation. Section 5 analysis the simulation results and the performance of our hardware predictor. Section 6 shows other related works. Section 7 discusses future works. Section 8 concludes the paper, and our references are listed in section 9.
2 BACKGROUND

The need for atomic sections (sections executed by only one thread at a time), is expressed by the need for locks, but there are many occasions where an atomic section is not actually guaranteed to conflict with another atomic section executing in the processor. The programmer surrounds such a section with locks because it might cause a problem if multiple threads execute it at once, but whether it actually conflicts with another atomic section or not isn’t something that the programmer can determine until execution time. Because the chance at incorrect execution is possible, locks are still included around these sections. However, there are alternatives, namely transactional memory. Transactional memory forms atomic sections into transactions, which can execute in parallel, then commit their memory operations at once. A few of the simpler implementations and the implementations of these two techniques relevant to our solution are discussed in this section.

2.1 LOCK AND UNLOCK

Lock and unlock operate to protect atomic sections by allowing only one thread to enter the atomic section at a time. How this is achieved is by the ownership of a lock. Threads normally compete for the lock until one successfully acquires it and enters the atomic section. Once it has completed the atomic section, it releases the lock through the unlock process. Lock implementations tend to fall into two different categories, spin and blocking locks. Spin locks continuously check a lock value until it is their turn to acquire, whereas blocking locks are able to switch to another thread to do other work while waiting on the lock [6].

Lock performance will serialize all atomic section’s execution, as shown in the diagram below. Most differences in lock implementations then differ in the acquire and release parts of the lock process, trying to reduce the wasted energy, such as moving away from spinlocks, or by ensuring more fairness by preventing starvation, such as Lamport’s bakery algorithm [5]. However, the exact choice of lock implementation does not affect our predictor, as it only relies on having a guaranteed safe fallback for atomic execution, which we will show in the implementation section. Here, we will talk about the simple lock implementation that is used in the parsec benchmark suite and built in to the pthreads multithreading library, pthread_mutex_lock [7][8]. Note, this lock falls into the latter variety of blocking lock implementations, though pthreads offers a spin implementation called pthread_spin_lock as well.
*Pthread_mutex_lock* is a blocking lock, included in the pthread library [7]. It is called by any program using the following two calls from within a thread:

**LOCK:** pthread_mutex_lock(address)

**UNLOCK:** pthread_mutex_unlock(address)

These two calls are normally within some lock and unlock *wrapper*, such that the exact implementation of the lock is hidden from the programmer to ease programming multithreaded applications. Therefore, the only calls a programmer typically uses, for example, in the *parsec* suite, is LOCK(address) and UNLOCK(address), where the address is the address of the lock. This is normally a pointer to some typecast integer or short that acts as the lock. Code using LOCK and UNLOCK will then normally use the following code structure:

LOCK(address);

*Atomic section code;*

UNLOCK(address);

This sort of implementation is simple for the programmer to use, and obscures the exact implementation of the lock from being a consideration while programming the atomic section. However, consider the situation where three threads are trying to execute the atomic section. Their performance is visualized in the graphic below:

*Figure 1: Lock Atomic Section Timing*

The lock then assures that no other threads are executing an atomic section at the same time, but a considerable amount of possible performance time is lost while waiting for other threads if the atomic sections could
be executed in parallel.

### 2.2 TRANSACTIONAL MEMORY

Unlike lock and unlock, transactional memory always tries to execute critical section in parallel, and can fallback or abort when a conflict is detected [3]. Each transaction is executed by a single process, and each process could try to do transactions at the same time. However, the transactional memory must still appear to have both serializability and atomicity [1]. The serializability property indicates that the steps of one transaction should never appear interleaved with steps of other threads. The committed transactions appear to execute in same consistent order by all processors. Atomicity means all modifications are indivisible and the transaction should either successfully commits all changes to memory instantaneously or aborts and undoes all changes.

When executing a lock-free critical section, transactional memory executes transactions in parallel. Compared to lock and unlock, the transactional memory greatly increases performance for critical section through this parallel execution. An example of transactional memory execution is shown below:

![Transaction Memory Atomic Section Timing](image)

According to version management and conflict detection, a transactional memory implementation can be categorized into eager version management/lazy version management and eager conflict detection/lazy conflict detection [2]. Version management is how a transactional memory handles new values in the transaction and old
values before the transaction. Eager version management puts the new value in place, copying old values to another location in the case of abort, while lazy version management leave the old value in place and writes the new values to another location, such as a queue. Conflict detection detects the overlay between write set of one transaction and the write set and read set of other concurrent transaction. Eager conflict detection is detecting conflict immediately upon a read or write, and lazy conflict detection normally waits until the commit to detect if a conflict has occurred.

Among various implementations of transactional memory, LogTM is eager version management and eager conflict detection. LogTM places new value in place and copies old values and transaction information to a log. LogTM also uses a sticky-M state to help detect conflicts when a block is evicted during a transaction. Conflict detection is eager, and traps to a conflict handler. Because LogTM uses this eager management, commits are very quick, but aborts are more costly as the log needs to be walked by the OS to restore old values. The following graphs visualize the transactional memory conflict detect and abort.

![Transaction Memory Conflict and Abort Timing](image)

**Figure 3: Transactional Memory Conflict and Abort Timing**

In a current commercial microprocessor, Intel designed TSX (Transactional Synchronization Extension) for use in their Haswell architecture. The Intel TSX hardware provides an eager version management and eager conflict detection model that provides a fallback instead of abort to a dedicated conflict resolver. Programmers could use Intel’s new TSX operations to write synchronization programs for concurrent and parallel applications, by using xbegin, xend, and xabort for the begin, commit, and abort commands.
3 Lock Elision Predictor

Since most effective TM implementations are optimized towards lowering commit cost, they suffer a higher abort cost if the transaction needs to abort. Some locked sections will conflict very frequently or always, meaning the need for both lock and unlock implementations along with transactional memory is needed to achieve the highest performance. The best choice would be to do lock and unlock on atomic sections where the abort chance is very high, and to do transactional memory when the abort chance is lower. However, this causes the programmer to choose between which one to do based on their program needs. This puts pressure on the programmer to know different possible implementations and to know the chance of abort for their atomic section themselves. With the other requirements for multithreaded applications, adding another consideration that could change drastically based on the different lock or transactional memory implementation is not desirable. To solve these issues, we propose the Lock Elision Predictor.

The lock elision predictor focuses three main objectives:

1. Keep implementation simple enough to be included in a wrapper, allowing the programmer to simply use the wrapper functions for LOCK and UNLOCK without needing to know whether locks or transactional memory is being used.

2. Make the implementation flexible enough that different lock implementations and different transactional memory implementations can be used without changing the predictor structure.

3. Design the predictor such that transactions are used as much as possible, then turn off and go to locks whenever a lock is highly likely to abort.

The first point eases the programmer’s effort for writing correct, effective multithreaded applications. If the predictor is simple enough such that any calls or necessary functions can be included in the wrapper, then the programmer can simply use LOCK and UNLOCK and be assured that the atomic section will execute correctly according to the current consistency model. In addition, the programmer doesn’t have to make the decision on whether to use transactional memory or locks themselves, as the predictor calls in the wrapper will choose the most optimal implementation while ensuring correctness.

The second point has to do more with the changing field for transactional memory and locks. There are many different algorithms available, and for our predictor to be as applicable as possible, it should not rely on one
particular implementation. Instead, the predictor should be flexible enough to be used with different lock implementations and different transactional memory implementations with very little to no modifications. This means that, as new or better implementations come out, or as available changes, the predictor can still be used. However, the predictor will still rely on two things. One, that the lock implementation is completely correct and will assure safe execution of the atomic section. Since this is normally the goal for lock implementations, this should not affect the predictor design. Second, the predictor will rely on the abort and commit operations of the transactional memory implementation. While the exact method for these operations differ among transactional memory implementations, their presence and meaning is all that is needed for the predictor. An abort means that a transaction has failed, and locks might need to be considered. A successful commit means that transactional memory should be considered again, as it has worked at least once.

The third point affects the actual design of the predictor the most. Since transactions can increase performance by such a large margin, then our predictor should bias itself towards trying transactions as much as possible. The point in which the predictor differs from an implementation of transactional memory such as Intel’s TSX is that TSX always tries a transaction, then has a safe fallback. The predictor tries to learn quickly if a lock has a high abort chance, then chooses to go with the safe fallback before any transactional memory actions take place. Not only does this save on abort cost, but depending on the transactional memory implementation, this could save time with initialization costs as well. For example, in Intel TSX, all values are copied before the transaction begins, which is additional time that could be saved before the abort would happen. However, such a confidence-like positive and negative case also needs a chance to become more positive (more likely to do transactions, in this case), and locks will never fail. Therefore, our predictor needs to have one more element because of this design point, and that is some method of resetting its confidence after a certain amount of time has passed to try and do a transaction again. A shorter time makes the predictor more flexible to short periods of aborts that pass quickly, while a longer time provides higher performance through avoiding transactions if the atomic section will always abort. Because of this, some sort of confidence counter with an exponential backoff that is initially biased towards transactions is our first choice for design.
4 Implementation and Evaluation

For testing our predictor, we worked inside the gem5 simulator. For this section, we will be discussing our proposed implementation for the predictor, and the details for our evaluation environment both inside and outside the simulator. We will start with a larger overview of the system, then talk about the needed modifications to each part to implement the predictor. For this section, we will also need to discuss the particular implementations we are building on top of. Our lock and wrapper implementation will be the implementation present in the parsec benchmark suite and inside the pthreads library [7][8]. Our transactional memory implementation will be the Intel TSX implementation in gem5 that was implemented by Pradip Vallathol, under the direction of Dr. David A. Wood [9]. We chose the lock implementation as it was the locks already present inside the parsec benchmark, and we chose the Intel TSX implementation in gem5 as it was the most recent transactional memory implementation inside the gem5 simulator. Below, you can see the layout of the system:

![Figure 4: General System Layout](image)

The main modifications will be within the lock and unlock wrappers, the implementation of the predictor within gem5, and the interaction of the predictor with the transactional memory implementation.

4.1 Lock and Unlock Wrappers

As shown in the background section, the pthreads lock and unlock are called from within a lock and unlock wrapper, generally using a LOCK(address) and UNLOCK(address) syntax. However, the Intel TSX implementation is not as generalized, and isn’t normally included in a wrapper. The general form for TSX code is as shown below:

```c
status = XBEGIN_STARTED;
if ((status = xbegin()) == XBEGIN_STARTED) {
```

// Atomic section code

} else {

  // Fallback code for abort or failure to begin

}

The address for the fallback code is normally saved at the xbegin call, so that if a failure or abort is called, the transaction knows where to go in the event of a conflict. The programmer then decides what to do in such a situation, whether to abort or to do some method of trying again, or another form of conflict resolution. This code is not nearly as flexible or easy to use for the programmer, as it is not within the lock and unlock wrapper. Therefore, to access the predictor, we join the two methods and add in an access to the predictor. Now, the code for LOCK(address) is as follows:

  status = xbegin(); // xbegin returns as normal, but now includes a call to the predictor.

  if (status == XBEGIN_STARTED) {

    return; // Transaction successfully started

  } else {

    pthread_mutex_lock(address); // Enter contention for lock since transaction failed

  }

Now, the return case in the event of an abort is set to the call to xbegin(), so that if something goes wrong, the predictor is accessed again. The predictor then always returns to lock after an abort call, so that the conflicting transactions will enter the lock and serialize. Since this code is now independent of the atomic section itself, it can be easily included in the lock wrapper itself. Since the accesses also use existing assembly instructions, no new assembly needed to be introduced at this level.

The unlock wrapper is even simpler. Since the fallback address is now at the point where the predictor is accessed, we can freely call xend. Xend does not commit if a transaction has not begun anyways, so the call is safe whether we actually started a transaction or not. In addition, the call to pthread_mutex_unlock also simply returns if the lock is not actually owned, so it is safe to call if we just did a transaction as well. Therefore, the code for UNLOCK(address) is now:

  xend();
4.2 Predictor

The predictor access is then implemented by slightly modifying the macro-op for XBEGIN to include the access to the predictor. This access is made using micro-ops within the processor to allow time to access the predictor in the execute stage. These micro-ops are placed above the beginning of the transaction itself, so that if the predictor chooses to do a lock, the entire cost of initializing the transaction can be avoided.

The predictor itself is a table of bit vector histories for that particular lock address. On an access, the address is used to generate an index in to the history table, and then a choice is made based upon the returned bit vector’s value, as shown below.

The history acts both as a confidence measurement and a way to measure the time until reset. If the last bit of the history is 1, then that means this lock has a strong bias towards transactions. We want to predict a transaction in this case, so our predictor will return, and the transaction code will begin. On a successful commit of the transaction, another 1 will be shifted into the bit vector. If an abort happens before the commit, however, a 0 will be shifted into the bit vector and a lock will be returned, to force the conflicting transactions that caused the abort to serialize. This is also the default case that all histories are initialized to, as we wish to bias towards transactions as much as possible.

If the last two bits are 10, however, that means we are in the weak transaction state. We have had one abort or are just resetting from doing locks, and we wish to try transactions once more, so we return transactions. If the
transaction aborts, we shift one more 0 into the bit vector and begin our countdown to reset. If the transaction successfully commits, then we shift a 1 into the bit vector, bringing us back to strong transaction state.

Finally, if the last two bits are 00, then we are in the reset state. We return a lock immediately, then shift another 0 into the bit vector. We continue to do this until the entire bit vector is 0, upon which we instead shift a 10 into the bit vector, bringing us to weak transaction state to try a transaction again. By varying the length of the bit vector, we effectively increase the amount of time for a reset to occur. The states for this modified confidence counter is shown below.

![Figure 6: Saturating Counter State Diagram](image)

Other improvements and modifications could be made to this predictor, such as an exponential backoff, or using a fully associative cache instead of a history table with index. Since the amount of locks can be fairly low, a fully associative, very small cache could reduce the chance of aliasing on the index to histories. An exponential backoff would allow more flexibility to short reset times if only a couple transactions actually abort, while increasing the reset time when they continue to abort repeatedly.

### 4.3 Transactional memory implementation

The transactional memory modifications for our implementation is small, including only a few more microops to access the predictor before the normal transaction implementation as shown in the previous section. However, the other modification was to the return address in the case of an abort. For Intel TSX, we adjusted this return address to the beginning of the prediction instead of the fallback code. Since the predictor will immediately
return a lock on an abort, this will cause the lock implementation to happen instead. This is to preserve the ability to use the lock and unlock wrappers.

Another change is to include the calls to the update functions in the commit and abort implementations. This is also fairly simple, as a couple micro-ops adequately allow time to call the predictor and update whether the transaction aborted or was committed successfully. Because these changes all happen “outside” the actual TSX implementation and do not modify its normal behavior outside of starting the transaction and where to return to, this makes the predictor flexible to other transactional memory implementations.

5. Results

Our evaluation of our predictor is split into two groups. The first evaluation was our efforts to measure the performance of our implementation inside of gem5, and since this is the more brief section, we will discuss it first. The second evaluation was parsing addresses accessed inside of locked sections, as if they were running in parallel. Here, we measured how many times locked sections would actually conflict, and then measured the accuracy of various predictors based on if they would have correctly chosen to do a lock or a transaction for a given atomic section. For all of our gem5 simulations, we used the following benchmark settings and simulator settings, given in the table below:

<table>
<thead>
<tr>
<th>Simulator:</th>
<th>Gem5 fs mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>OS</td>
<td>CentOS x86 64 bit</td>
</tr>
<tr>
<td>Memory Size</td>
<td>3GB</td>
</tr>
<tr>
<td>Protocol</td>
<td>Modified MESI for TSX</td>
</tr>
<tr>
<td>Benchmark suite</td>
<td>Splash2x from Parsec 3.0</td>
</tr>
<tr>
<td>Test size</td>
<td>Small sample size</td>
</tr>
<tr>
<td>Processor Count</td>
<td>8</td>
</tr>
<tr>
<td>Lock implementation</td>
<td>Pthread mutex</td>
</tr>
<tr>
<td>TM implementation</td>
<td>Intel TSX</td>
</tr>
</tbody>
</table>

*Table 1: Simulator and Benchmark Settings*
5.1 Full System evaluation

The first step in our evaluation was to measure the performance of benchmarks in their current lock implementation. We then modified gem5’s simulator to allow us to measure the non-idle cycles spent inside of atomic sections, and took the longest measurement, subtracting the rest from the total. This was to get a rough calculation of how fast a perfect transactional memory implementation would speed up our benchmarks, as all threads would be able to execute the atomic sections in parallel. The results of this evaluation is given below. The results are normalized to the lock case, showing the reduction in busy cycles when doing perfect TM:

![Normalized Base and Perfect TM Busy Cycle Count](image)

*Figure 7: Perfect TM Busy Cycles vs Lock Busy Cycles*

We then attempted to evaluate our implementation for regular transactional memory, regular TM with a predictor, and a perfect predictor. Unfortunately, our evaluation ran into a severe problem with the memory implementation for Intel TSX in gem5. Because of this, we do not have final statistics from full system emulation in gem5. However, we were able to still gather address traces and perform predictor accuracy analysis, as given in the Log Trace Evaluation section below.

5.2 Log Trace Evaluation

We experimented with a few adjusting knobs to tune our predictor and determine how each control affects the prediction results. We first added m5 calls in gem5 to print the current clock cycle, thread id, and whether the
instruction is a lock, dataAccess, or unlock. Next, we instrumented the benchmark source files to access the lock and data before actually locking, which removes exclusion and allows multiple threads to conflict. Last, we wrote a program to parse the log traces to determine which critical section would have conflicted and how each predictor would have behaved. The four predictors we have are 1) always predict a transaction, 2) our current hardware predictor implementation with a 2 bit saturating counter and try transactions again after 5 locks, 3) a 1-bit saturating counter, and 4) a 2-bit saturating counter. Both the 1-bit and 2-bit saturating counter uses the trace as an oracle to know whether the transaction would have been successful while our current hardware makes prediction only based on previous transactions.

We first instrumented the oceans benchmark and ran it with 2 threads and various input sizes as shown below. Since there are only 2 threads, the probability of conflicting within the critical section is low for large enough grid size, which makes prediction easy.

![Figure 8: Prediction on Ocean with 2 Threads](image)

Figure 8: Prediction on Ocean with 2 Threads. An fast way to skim the data is that dark green is good (predicted transaction successfully) and dark red is bad (predicted transaction unsuccessfully) while light green and light red is predicted lock, which does not have an effect on speedup. The four different pattern fillings matches the four predictors from left to right respectively.

Next, we ran the oceans benchmark with 4 threads as shown below. Since there are more threads and the workload is distributed evenly, there is a higher probability of conflicts for small grid sizes. Conflicts always occur if the grid size is small enough as each thread have very little work to do before reaching to the critical section.
together, making it easy to predict lock. On the other hand, even though workload is evenly distributed, large grid sizes barely have any conflicts because some threads will likely straggle behind other threads as the workload increases. Therefore, conflicts rarely occur if the grid size is large enough, making it easy to predict transactions.

Likewise, prediction is more difficult on an intermediate grid size of 10x10 as shown in the figure below.

![Figure 9: Prediction on Ocean with 4 Threads](image)

Next, we instrumented the critical sections of other benchmarks in splash2x and ran them through gem5 to produce a trace to parse. The figure below shows the results of the testbenches we were able to instrument and run successfully with various input sizes and various number of threads.

![Figure 10: Prediction on splash2x benchmarks with various inputSizes and numThreads](image)
With the exception of fft, the results above confirms findings of related works [2] that transactions are successful about 98% of the time in splash 2x benchmarks. fft uses locks only upon worker creation, so there were only 2 critical sections with 2 threads and 4 critical sections with 4 threads. fft does use barriers for synchronization, which was not implemented on top of locks in parsecmgmt (PARSEC benchmark management). radix, lu_continousBlocks and lu_nonContinousBlocks behaves similarly as fft with very few critical sections.

Speedup for doing a predictor can be increased if the transaction predictor is correct and if the critical section contained false sharing. However, speedup is decreased if the transaction predictor is incorrect, which requires cleanup time for aborting on top of restarting the critical section. The speedups can be estimated as $numLocks \times %predictTransaction(speedupOfTransaction \times successRate - slowdownOfFailedTransaction \times failureRate)$.

Predicting to perform a lock will have no impact on speedup because that is the base case anyway. Although not calculated, the speedup is negligible in these splash2x testbenches because they are already optimized to avoid false sharing. Therefore, our results shows only performance degradation because all benchmarks in the splash2x test suite is already optimized to avoid false sharing. We do expect to see speeds on less optimized testbenches with false sharing.

6 Related Works

Speculative lock elision [1] is a hardware implementation of removing false sharing between two critical sections. The transaction or lock is dependent on a confidence metric. This work expands on the idea of speculative lock elision by focusing on the prediction techniques and tradeoffs, while building on behavioral patterns of other transactional memory findings, such as the low abort rate findings of LogTM.

LogTM [2] is an implementation of transactional memory with eager version management and eager conflict detection to speedup commits and reduce wasted work. We used two ideas from this work: One, the idea that transactions are successful 98% of the time, and two, optimizing the benchmark to eliminate false sharing gives 1.58 speedup to form the basis of why implementing a hardware predictor will be successful. However, our predictor focuses on determining whether to use transactions or locks as opposed to making transactions faster, therefore, our solution could be used in conjunction with LogTM for possibly more performance gain.
7 Future Works

Future works includes borrowing and experimenting with more ideas from conflict avoidance and branch predictor. In particular, we could do exponential backoff before trying transactions again, fine tune the predictor to use the optimal number of saturating bits, implement multiple predictors to capture multiple behavior patterns and then implement a tournament predictor on top. In addition, working with a full system emulation of multiple transactional memory implementations would allow exploration of how different techniques associated costs might affect overall performance when coupled with the predictor.

8 Conclusion

The transactional memory predictor is a promising approach to provide hardware prediction for atomic sections in a way that is more flexible and in tune with transactional memory behavior than alternative solutions. The transactional memory predictor is optimized to provide high prediction accuracy with low hardware cost and ease of programmer use. While there is more exploration to be done in the performance of our predictor with benchmarks both tuned to avoid possible conflict and benchmarks that aren’t as tuned, the potential use for more flexible applications is still present from the programmer’s standpoint and the ease of incorporation with different transactional memory implementations.

More full system emulation and examination on other benchmarks are still needed to measure actual speedup, as our metrics currently do not predict much speedup with outside address analysis. However, the accuracy of predictors to the current behavior does still suggest that, with other full system tests and further customization, improvement is still possible. Furthermore, since the implementation of our transactional memory predictor is flexible enough to allow this adjustment to actual behavior, it still offers promising performance and flexibility with a wider range of benchmarks and transactional memory implementations.
9 Appendices and References


