Hybrid Transactional Memory System Based on Two-Level Adaptive Prediction

Abstract

Transactional memory has quickly been integrated into mainstream parallel computing paradigm such as intel's TSX and IBM's Blue Gene/Q. However, it is still controversial as to when to choose between transactional memory and locks. In this paper, we propose a new hybrid transactional memory system that dynamically adapts its execution between transactional memory and locks based on a novel two-level adaptive prediction strategy. We fully implement our scheme in RSTM. We also describe and evaluate possible hardware implementations. Real hardware and software measurements show that as compared to a traditional software transactional memory algorithm LLT, our strategy achieves an average speed of 17.1% for STAMP applications with high contention rate.

1. Introduction

Transactional memory (TM) has long been known as an effective approach to mitigate the performance inefficiency and ease the programming burden of software locks. After the trough of disillusionment of tremendous academic proposals, TM is climbing on the slope of enlightenment as the industry finally adopted this programming paradigm. Implementations such as intel's Transactional Synchronization Extension (TSX) [11] and IBM's Blue Gene/Q [12] exemplified so.

Among various transactional memory proposals, hybrid approaches combining the programmability of software transactional memory (STM) and performance of hardware transactional memory (HTM) are most widely used. Both TSX and Blue Gene/Q fall in that sort. Similar to STM, hybrid approaches encode transactional memory semantics in software libraries (and runtime system), and expose simple APIs to programmers; therefore they only require mild source-level change. Meanwhile, they employ relatively simple hardware structures to accelerate critical STM execution paths without drastically changing the microarchitecture and ISA, and therefore are more easily to deploy in practice.

However, transactional memory is not a panacea. The performance of transactional memory on a simple benchmark `CounterBench` [2], in which each thread increments a global counter in each critical section, is orders of magnitude worse than a simple lock based implementation. Albeit extreme, the `CounterBench` showcases situations where traditional locks are beneficial over transactional memory. In complex applications, even with a priori knowledge of the runtime behavior of critical sections, it is next to impossible and insufficient to statically to decide between the two.

To this end, we aspire to ask a fundamental question: how to effectively take advantage of both locks and transactional memory? In this paper, we approach the answer by proposing a new hybrid software transactional memory scheme that dynamically predicts and adapts itself between transactional memory and locks at runtime. By shifting the decision to the transactional memory system, programs quickly react to runtime behaviors while easing programmers' efforts.

Among others, our key insight is to stall a thread before entering a transaction if it has a very high chance to abort later on. We employ a two-level prediction scheme to predict the likelihood of abort based on both the abort rate information and pairwise conflict history. At the first level prediction, if the abort ratio of a transaction is below a certain threshold, it is predicted to be unlikely to abort and therefore is allowed to enter the transaction for speculative execution; otherwise, the second level predictor reviews the abort history of the transaction and stalls it if any transaction that aborted the current transaction before is still active.

Measurement on real system show that for applications with high transaction abort rate, our strategy is able to outperform a traditional software transactional memory algorithm LLT by 17.1%. We also evaluate possible hardware implementation via modeling. Promising modeling results show that implementing our strategy in hardware will achieve even better results.

The rest of the paper is organized as follows. We introduce the background of hybrid transactional memory and related TM algorithms in Section 2. Section 3 details the design of our proposed scheme. We evaluate it in Section 4, followed by analytically modeling the hardware implementation of our scheme. We discuss a few interesting issues in our design and implementation in Section 6. We finally conclude in Section 7.

2. Background

2.1 Improving STM Efficiency

Transactional Memory simplifies the task by supporting the definition of groups of load/store instructions that are considered as a transaction to have atomicity, consistency and isolation requirements. There are two basic types of transactional memory: Hardware Transactional Memory (HTM) and Software Transactional Memory (STM). HTM usually makes extensions on ISA, with some special transactional operations. It has lower overhead (especially for conflict detection), provides strong encapsulation, and possibly decreases power. But it is really complicated to design and thus not yet available. On the contrary, STM is much more common for transactional memory implementation and naturally more flexible and easier to change. Therefore, STM is currently the more widely used approach in transactional memory design. However, STM only solves the problem of programmability (compared with fine-grain lock) while suffers from larger overhead and thus not efficient in performance and power.

In order to solve the inefficiency problem, a lot of research has been done in accelerating STM, among which, hybrid TM and adaptive STM are representatives. Hybrid TM leverages the merits of both hardware and software. One way of implementing hybrid TM is bringing hardware structure (ASIC) into STM system to accelerate transaction procedures, and the other makes the system switch between HTM and STM based on program phase. Adaptive STM comes into being from a fact that different STM algorithms and scheduling methods outperform in different environments. For example, some algorithms (eager acquisition and eager versioning) take aggressive speculation, giving them high efficiency when little conflicts between transactions happen but larger abort overhead; some algorithms are more conservative and even serialize all transactions, but have better performance if conflicts are overwhelming. By switching between different algorithms/scheduling, adaptive approaches try to find the best efficiency in different program phase indicated by some monitoring mechanism.

2.2 Related Work

In this section, we review a few related work as to accelerating STM.

Worst-Case Progress Many STMs support a “Serial Irrevocable” (SI) mode, where only one transaction is executed at a time. This mode has the lowest conflict overhead but does not leverage any possible parallelism. In some papers, STM algorithm will be changed to SI mode (Coarse Grained Lock - CGL algorithm in RSTM system) after a thread encounters a big number of consecutive abort. A simple example for this situation is several threads are modifying a same counter [4, 5].

Phased Execution PhTM switched between hardware and software modes on a machine with hardware TM support, according to run-time information, including transaction type (supported in hardware or not), number of consecutive aborts and non-transactional work time. But this method did not do anything about switching between different STM algorithms [6].

Selecting Locks or Transactions This method utilized both static and dynamic analysis to make choice between lock and STM. Pure lock is sometimes the best when a conflict will probably happen or when no parallelism can be exploited (for instance, only one thread is being executed) [7].

Pathology Avoidance The system selects from 10 different STM algorithms and decisions are based on algorithms’ possibility of pathology and precision of conflict detection. Switching algorithms at run time is supported in RSTM system but all threads need to use the same algorithm and synchronization has to be applied [8].

Applying Machine Leaning Since each of STM algorithms appears to be suited to certain

workloads and architectures, the best algorithm is selected when running the program based on dynamic profiling information and also using machine learning techniques [9].

Proactive TM Scheduling One paper proposed a RELSTM system which predicts future conflicts between transactions and then adapts the scheduling (backoff or serialize) according to the potential conflict types [10].

2.3 STM Algorithms

There are many existing algorithms for STM. They differ in two fundamental concepts in TM algorithms: write-acquisition and write-versioning [2].

We choose to use LLT (Lazy-write-acquisition, Lazy-write-versioning, Timestamp) as the algorithm throughout the entire paper. LLT is a simple and elegant algorithm with low overhead. There is a global clock that transactions read when they start. Each memory location contains a data structure (owner-record) that records the last time that a location was modified. If a transaction encounters a location that was written after it starts, it assumes it is inconsistent and aborts and retries. At commit time, the transaction acquires all of the owner-records associated with locations it is modifying, checks to make sure that all of the locations it read still have a timestamp earlier than its start time, increments the global time, and then writes out all of its updates and modifies the associated orecs to be the new time.

3. Two-Level Adaptive Hybrid Transactional Memory

In this section, we first provide a high level overview of our hybrid TM design, with a focus on two-level adaptive prediction scheme (Section 3.1). We then describes both levels in greater details (Section 3.2, 3.3).

3.1 Design Overview

Different from most other TM algorithms which always allow a thread to enter the transaction for speculative execution, we argue that if a thread will eventually abort in the future, it is better to stall it before potential conflicts are resolved. Otherwise the thread is just occupying the computation resources and consuming power for the work that will eventually be abandoned later. In reality however, precisely making such decision before entering a transaction is most often impossible, and the best-effort overhead of tracking related information may outweigh the benefits.

Our two-level adaptive predictor sets to make a trade-off between accurate prediction and prediction overhead. It quickly makes the first level decision simply based on the abort rate of a transaction. If a transaction is most often able to commit in the past, it is very likely to commit successfully next time. The first level prediction provides fairly accurate results with negligible overhead (Section 3.2). On the other hand, even if a transaction has been aborted mostly in

previous executions, it may not necessarily abort this time if no conflicting transactions are active during its current execution. The second-level predictor makes such predictions by keeping track of previous conflict history of each transaction. It more accurately predicts the chances of abort with higher bookkeep overhead. Therefore it is important to carefully design the second-level predictor with lower the overhead but without sacrificing the accuracy (Section 3.3).

3.2 Abort-Ratio Based Level 1 Prediction

The goal of the first level predictor is to quickly decide if it is beneficial to use transactional memory with very low prediction overhead. We make such decision using the abort ratio of each transaction on a per-thread basis.

We maintain a global abort history table (AHT) which contains an entry for each <thread, transaction> tuple. Each time a thread (tid) is aborted on a transaction (txid), the corresponding entry <tid, txid> is incremented. The ratio of the abort counts over the total commit counts denotes how often a particular transaction in a particular thread is aborted before. We use the abort ratio to indicate whether a transaction will be aborted in the future.

The overhead of maintaining AHT is negligible since each thread has sole control of each entry without worrying about race condition. It is also possible to have all the threads share the same abort counter for each transaction to lower AHT memory requirement. However, there are two major drawbacks of it. First, it requires mutual exclusion to access the counter since multiple threads share the same counter. Second, multiple threads may exhibit different behaviors (such as control flow divergence) on the same transaction, leading to different abort/commit results; sharing the same counter across all the threads pollute the accuracy.

3.3 Pairwise Conflict-Aware Level-2 Prediction

Complementary to level-1 predictor, level-2 predictor sets to decide whether a thread should enter a transaction even if its abort ratio is high. A often aborted transaction should still be allowed to enter the transaction if it does not abort this time. Level-2 predictor predicts so based on a simple hypothesis:

Hypothesis: Two transactions are very likely to conflict with each other as long as they conflict once. There is a set of conflicting transactions (conflicting set) for each transaction.

Based on the hypothesis, level-2 predictor tries to guarantee for each thread that there is no transactions in the conflicting set being executed when it enters the transaction. We propose and evaluated three different schemes: brute force searching, simple hashing (Section 3.3.1) and least recently conflicted (Section 3.3.2).

3.3.1 Brute Force Searching and Simple Hashing

Brute Force Searching The brute force approach is to maintain a precise record of the conflicting set for each transaction, and to check the status of all transactions in the conflicting set before entering the transaction.

Brute force matching requires two global structures: a conflict history table (CHT) and an active transaction vector (ATV). Each transaction has an entry in the CHT; each entry is a vector, with each vector element recording the number of times a particular transaction aborts the current transaction. For example, the n^{th} element in the m^{th} entry containing 5 indicates that transaction `n` has aborted transaction `m` 5 times. Each transaction also owns an element in the ATV, recording the number of threads that are currently executing the transaction (active count).

Before entering a transaction, we first check the its corresponding entry in CHT, and stall the thread if there is any conflicting transaction currently still active, i.e. the active count is not 0 in ACV.

Although most accurate, in implementation, the brute force approach has to check through all the elements in a CHT vector, leading to unaffordable memory and computation overhead, especially considering there might be hundreds of different transactions in an application.

Simple Hashing A simple hashing approach can help tackle the overhead issue of brute force method by mapping multiple different transactions to the same vector element in the CHT entry. It effectively limits the size of each entry in CHT and therefore can greatly reduce the search overhead.

The major drawback of this simple hashing strategy is false positive, since multiple transactions might be mapped to the same element in the CHT entry. For example, it is possible that transaction `m` and transaction `n` are both mapped to same element in the CHT entry for transaction `k`, while only transaction `m` conflicts with transaction `k`, forcing transaction `k` to stall unnecessarily if transaction n is active. Mistakenly serializing transactions could possibly hurt the performance.

3.3.2 Conflict Cache

We propose conflict cache to reduce the searching space while does not incur false positive. Instead of searching all the transactions (i.e. all the elements in a CHT entry), we only search the most recently conflicting transactions.

Cache Structure Conflict cache is a set associative cache indexed by the transaction index. Each set contains the most recently conflict transactions that previously aborted the indexed transaction. For example in Figure 1, T0 was previously aborted by T1, T3, T7 and T10.

Index	Data			
T0	T1	T3	T7	T10
T1	T2	T3	T6	T0
T2	T3	T7	invalid	invalid
T3	T1	T3	T27	invalid

Figure 1. Conflict Cache Structure

Instead of checking all transactions as in brute force strategy, conflict cache allows us to only check a small number of transactions in the cache structure before entering each transaction. Meanwhile, it does not have the false positive issue as in the simple hashing strategy since only conflicting transactions are recorded in the conflict cache.

There might be a false negative issue of employing conflicting cache because it potentially neglects conflicting transactions not currently in the conflict cache. Depending on the associativity of conflict cache, we can achieve different coverages of conflicting transactions.

Replacement Policy We propose the least recently conflict (LRC) replacement policy. Similar to Least Recently Used (LRU), LRC maintains a rank among all ways in the same set. After an abort, LRC policy will choose the way with the lowest rank as a victim, replace it with a new transaction number and promote it to highest rank (rank 0). If the conflicting transaction already exists in the set, LRC will simply promote it. As an example, assuming a conflict cache started with Figure 2.

Index	Data				Rank			
T0	T1	T3	T7	T10	0	1	2	3

Figure 2. Cache Snapshot A

if T0 was aborted by T2, conflict cache would be updated to Figure 3.

Index	Data				Rank			
T0	T1	T3	T7	T2	1	2	3	0

Figure 3. Cache Snapshot B

later on T0 was aborted by T1, the snapshot of conflict cache status would be like Figure 4.

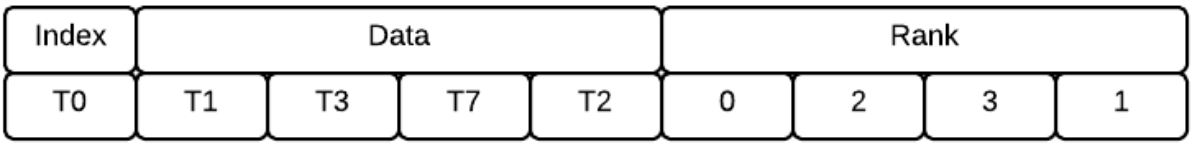

Figure 4. Cache Snapshot C

4. Experiments and Evaluation

4.1 Methodology

We fully implement our two-level adaptive prediction scheme into RSTM [2]. RSTM is a highly modular STM implementation containing various algorithms implementations. We perform evaluation on the STAMP benchmarks [1]. STAMP is a standard transactional memory benchmark covering a variety of algorithms and domains. The characteristics cover a wide range of transactional memory behaviors such as varying degrees of contention, short and long transactions, and different sizes of read/write data sets. Due to the default bug of RSTM, we are not able to run `laybrith` and `yada`. All input parameters are recommended in [1], using the large input sets. We also run two inputs for both `kmeans` and `vacation`.

We implement all our modifications in `STM_BEGIN` and `STM_END` macros in RSTM. Overall, our modifications do not require changing application source code. To guarantee safe concurrent accesses of global data structure, such as AHT and conflict cache, we carefully design fine-grain locks for these global data structures.

In our experiments, we explore design space of different parameters in our heuristics and find the following set of parameters most effective: 1) 10% for the threshold abort ratio for level one predictor; 2) size of each CHT entry is 10, and 3) set associativity of conflict cache is 8.

4.2 Results and Analysis

We performed all the experiments on a 4-socket 24 core Xeon X5660 server running 256 threads. Figure 5 shows the overall performance of various transactional memory algorithms, normalized to LLT.

The brute force strategy generally performs much worse than LLT, indicating the monitoring overhead outweighs the performance gain of reducing aborts. Simple hashing and pure conflict cache (LRC) based strategies greatly alleviate the monitoring overhead, but are still not able to compete with LLT. The two-level adaptive strategy outperforms LLT in 6 out of 8 applications due to much lower abort rate, as indicated in Table 1.

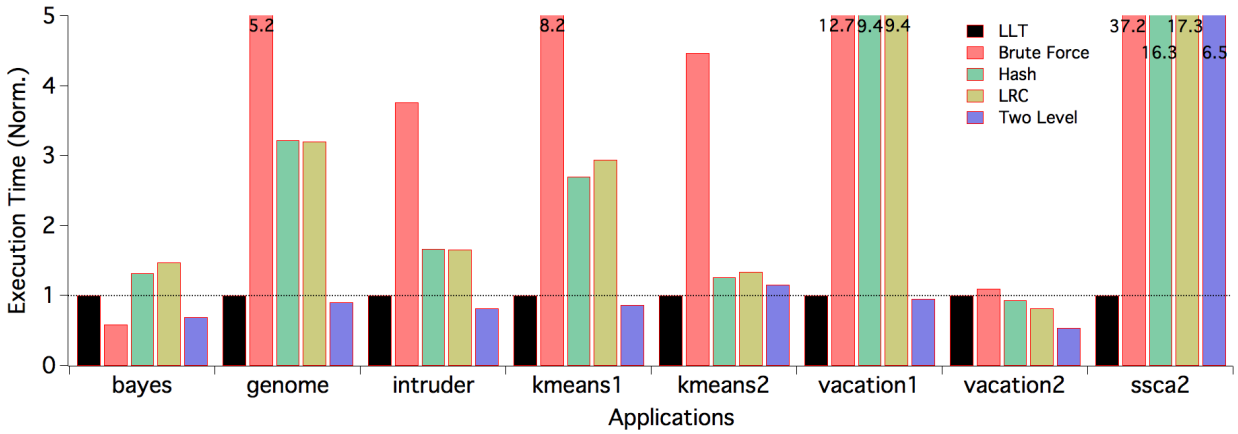

Figure 5. Performance comparison of various transactional memory algorithms, normalized to LLT. LRC denotes the pure conflict cache strategy without two-level prediction

We do notice a significant outlier application `ssca2` where even the two-level prediction strategy is still 6.5X slower than LLT. This is because almost all the transactions in ssca2 are short transactions that have very low contention rate. In this condition, most of the execution time spent in prediction is wasted, leading to terrible performance.

Table 1. Abort rate comparison between the 2-level prediction strategy and LLT.

	bayes	genome	intruder	kmeans1	kmeans2	vacation1	vacation2	ssca2
LLT	**79.72%**	**2.51%**	**1004.75%**	**98.24%**	**388.27%**	**0.87%**	**6.641%**	**0.0067%**
2-level	**3.39%**	**0.47%**	**8.01%**	**7.31%**	**8.91%**	**0.49%**	**1.56%**	**0.00043%**

5. Hardware Acceleration

In this section, we describe how previous software implementations can be implemented in hardware for acceleration. We modestly extend both the ISA (Section 5.1) and the microarchitecture (Section 5.2) to support our new hybrid transactional memory system.

5.1 ISA Support

We extend the ISA with 2 new instructions: `TM_Cond_Enter` and `TM_End`. The programming model is as simple as following:

```
TM_Cond_Enter()
some TM transaction work...
TM_End()
```

`TM_Cond_Enter` hashes the current PC to a transaction ID to index various hardware tables (AHT, CHT, conflict cache etc.) depending on which level-2 prediction scheme to use. If the transaction is scheduled to execute (speculatively) according to our prediction, ATV will be updated to indicate that this transaction becomes active and the thread continues executing the rest instructions in the transaction. If a transaction is scheduled to stall, a conditional branch will point the control flow to the `TM_Cond_Enter` instruction again.

`TM_End` indicates that a transaction has successfully committed from the STM's perspective. Therefore this instruction will update the ATV since current TM is no longer active.

Theoretically, we also need to modify the semantics of memory instructions within transactions to update conflict cache, AHT and CHT upon transaction abort. However, these modifications can be realized with slight microarchitecture enhancement and therefore do not require ISA change.

5.2 Microarchitecture Support

To support the instructions mentioned above, we enhance the microarchitecture with following additional hardware structures (also shown in Figure 6):

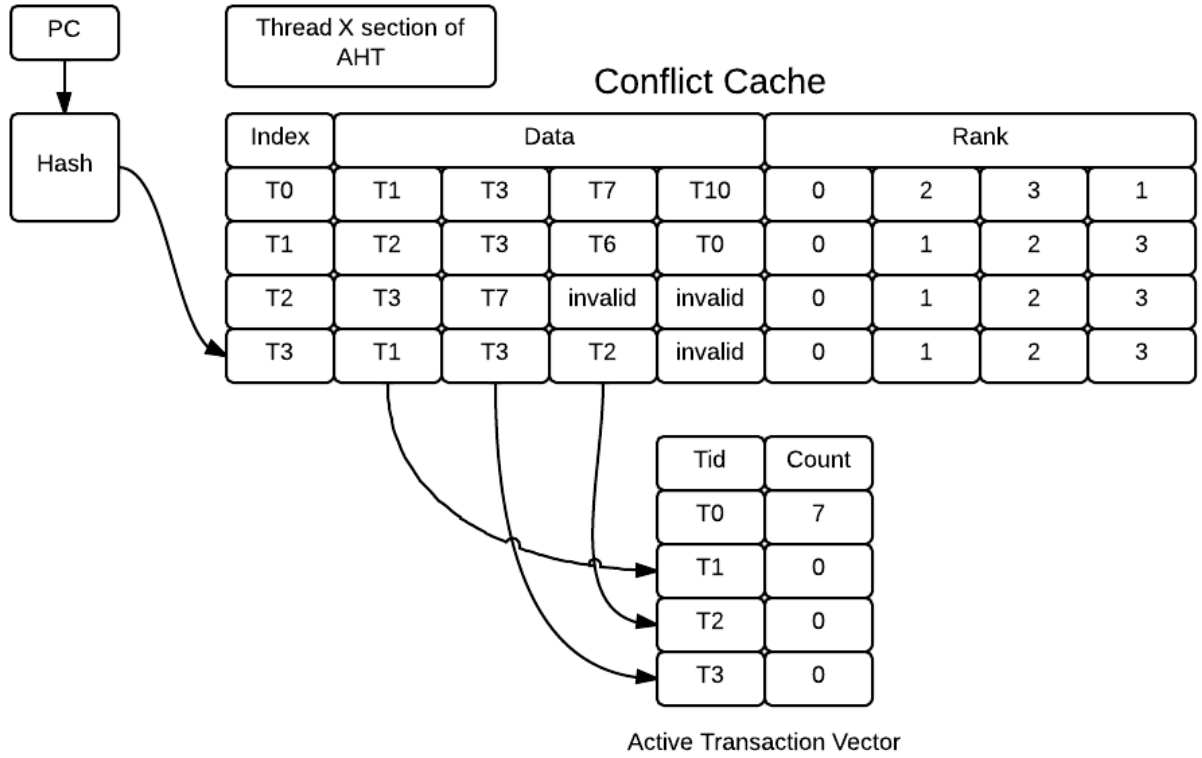

Figure 6. Microarchitecture of Prediction Accelerator

1. Hash logic that converts a transaction's PC into its transaction ID.
2. AHT containing abort counters for each transaction in each thread.

3. ATV that keeps track of current active transactions among all threads.
4. Conflict cache.

5.3 Timing, Area, and Power Estimation

The biggest overhead in our hardware implementation is the conflict cache. We use CACTI 6.5 [3] to estimate the timing, area and power of it. Table 2 lists the configuration used for evaluation.

Table 2. Conflict cache configuration.

Technology	32 nm, all itrs-hp
Operating temperature	350 K
Access mode	Fast
Cache size	1024 byte
# of transactions in one set	8
# of sets in conflict cache	128
Width of transaction id	8 bit

Table 3 shows that the area and power overhead is negligible as compared to the state-of-the-art microprocessor. Most importantly, the delay time shows that it is possible to finish conflict cache operation in one cycle.

Table 3. Area, power and timing of conflict cache.

Total area	0.002716 mm^2
Total dynamic energy / access	0.0015 nJ
Total leakage power	0.410359 mw
Delay with output driver	0.16 ns

6. Discussion

Why does abort ratio help first level prediction ?

When doing speculative execution for transactional memory, abort ratio is a very important factor that represents the behavior of the running application. When abort ratio is high, it means that transactions have a lot of conflicts and also the work of transaction occupies a large percent of the whole application. Based on this, we could get information about what type of application we are running as well as TM-related characteristics we are currently have, and then predict the

possibility of the transaction's abort. Previous research used abort number to adapt STM policies but they used the same counter for all transactions in the same thread. In our approach, we dynamically profiled those information for each transaction in each thread, which we believe is more accurate.

In same thread, different transactions still have different abort possibilities and sharing the same abort ratio ignores some information. For example, transaction A has a very large read set and long running time while transaction B is very small only containing a single read and a single write operation. When they are in the same thread, with the same history abort ratio for all transactions, it is obviously that their abort rate can be quite different. The smaller one has less possibility of conflicts with others. Therefore, we believe our criterion in level 1 is more accurate and beneficial for predicting transaction abort.

Why and when is the conflict history useful for second level prediction?
In our method, we believe that if transaction A aborts transaction B before, the conflict would lead to another abort very probably if A and B are active simultaneously in future. However, this might not be true in some cases.

First, there can be control flow and value divergence in a transaction, which means the writing set and reading set of a transaction are not fixed. Therefore, previous conflict between two transactions might not result in conflict again. Secondly, interleaving of transactions from different threads vary all the time, which affects the abort also. For example, transaction A aborts transaction B because transaction A commits a write before a read operation in transaction B. However, next time when both transactions are active, it is possible that transaction A starts much later and the write operation which cause the abort before will not commit before transaction B commits.

Issues described above have impact on our prediction scheme. However, we believe our hypothesis will still hold true according to our experimental results.

7. Conclusion and Future Work

This paper is our preliminary work towards answering a fundamental question in transactional memory: when is beneficial to choose between transactional memory and locks? We show that a dynamic adaptive scheme based on a two-level prediction strategy is promising and achieve an average speed up 17.1% on STAMP benchmark applications with high contention rate as compared to the LLT algorithm in RSTM.

We are exploring the following research directions. We are conducting design space exploration on the parameters on both level 1 and level 2 predictions. It is also interesting to analyze the overhead of abort in some applications to make a better scheduling for transactions.

8. References

[1] Chi Cao Minh, et. al., STAMP: Stanford Transactional Applications for Multi-Processing. In Proc. of IISWC 2008.

[2] Michael L. Scott et al. Lowering the Overhead of Nonblocking Software Transactional Memory, In Proc. of TRANSACT 2006.

[3] S. J. E. Wilton and N. P. Jouppi, "CACTI: An enhanced cache access and cycle time model," IEEE J. Solid-State Circuits, vol. 31, no. 5, 1996.

[4] Y. Ni, et. al., Design and Implementation of Transactional Constructs for C/C++. In Proc. of the OOPSLA 2008.

[5] A. Welc, et. al., Irrevocable Transactions and their Applications. In Procd. SPAA 2008.

[6] Y. Lev, et. al., PhTM: Phased Transactional Memory. In Proc. of TRANSACT 2007.

[7] T. Usui, et. al., Adaptive Locks: Combining Transactions and Locks for Efficient Concurrency. In Proc. of PACT 2009.

[8] M. Spear, Lightweight, Robust Adaptivity for Software Transactional Memory. In Proc. of SPAA 2010.

[9] Q. Wang, et. al., Towards Applying Machine Learning to Adaptive Transactional Memory. In Proc. TRANSACT 2011.

[10] David Sainz, et. al., RELSTM: A Proactive Transactional Memory Scheduler. In Proc. TRANSACT 2013.

[11] Intel's Thread Synchronization Extension (TSX): http://en.wikipedia.org/wiki/Transactional_Synchronization_Extensions

[12] Amy Wang. Evaluation of Blue Gene/Q Hardware Support for Transactional Memories. In Proc. of PACT 2012.