2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing

Improving Performance of Transactional Applications through Adaptive Transactional Memory

Thireshan Jeyakumaran, Ehsan Atoofian, Yang Xiao
Lakehead University
Thunder Bay, Canada
{tjeyakum, atoofian, yxiao4}@lakeheadu.ca

Zhen Li, Ali Jannesari
Technical University of Darmstadt
Darmstadt, Germany
{li, jannesari}@cs.tu-darmstadt.de

***Abstract*—Transactional memory (TM) has become progressively widespread especially with hardware transactional memory implementation becoming increasingly available. In this paper, we focus on Restricted Transactional Memory (RTM) in Intel's Haswell processor and show that performance of RTM varies across applications. While RTM enhances performance of some applications relative to software transactional memory (STM), in some others, it degrades performance. We exploit this variability and present an adaptive system which is a static approach that switches between HTM and STM in transaction granularity. By incorporating a decision tree prediction module, we are able to predict the optimum TM system for a given transaction based on its characteristics. Our adaptive system supports both HTM and STM with the aim of increasing an application's performance. We show that our adaptive system has an average overall speedup of 20.82% over both TM systems.**

Keywords- Restricted Transactional Memory; Software Transactional Memory; Decision Tree; Performance

I. INTRODUCTION

Transactional Memory (TM) is becoming increasingly popular for developing parallel applications for multi-core processors. TM provides programmers with an atomic construct (transaction), which can be used to guarantee atomicity of accesses to the shared variables. This is much simpler than conventional lock-based programming, which requires significant programming effort to avoid synchronization bugs such as deadlock, livelock, and priority inversion. In a TM program, the complexity of enforcing atomicity of shared variables is delegated to the underlying system, instead of any hand crafted synchronization schemes defined by the programmer. Transactional memory comes in two variants: Software Transactional Memory (STM) and Hardware Transactional Memory (HTM).

STM is the simplest approach of transactional programming in which all transactions are implemented entirely in software. There has been intensive research done on practical implementations of STM which has lead to the development of many systems such as TL2 [1], TinySTM [2], and CTL [3]. All of these systems execute transactions using software-based resources. This includes conflict detection, consistency of transactional reads, preservation of atomicity, preservations of isolation, etc. This results in poor performance in some applications due to the overhead associated with initiating and overseeing the system.

To mitigate the overhead of STM, hardware manufacturers such as Intel [5] and IBM [7, 8] have incorporated hardware support for transactional memory in their respective chip multiprocessors. In HTM, speculative transactions are executed using hardware resources (such as internal buffers, caches, etc.). In 2013, Intel released the Haswell processor which incorporates hardware support for TM, called Restricted Transactional Memory (RTM) [5]. RTM exploits cache-coherence protocol [6] in order to track transactional conflicts. Yet, RTM is no magic solution as an alternative to STM due to its own set of limitations and constraints. RTM and other similar HTM systems such as IBM's BlueGene/Q [7] and Power8 [8] follow the 'best-effort' protocol, meaning it does not provide forward progress. In other words, there is no guarantee that a transaction will successfully commit in RTM; essentially requiring a fallback path to successfully execute an application in the event of an abort. Generally, a fallback path is an alternative software policy to guarantee successful execution. This software policy can be as simple as acquiring a lock and executing it non-transactionally. Another vital limitation of RTM is cache capacity where the success rate of a transaction depends on whether the data set (read/write) fits inside the cache. There are also other architectural constraints that limit performance in RTM such as context switching, interrupts, I/O instructions, etc. Thus, RTM is not a polished product as of yet, but there is potential for performance gain.

In this paper, the focus is on implementing an adaptive system that exploits both HTM and STM at transaction granularity. The goal is to achieve performance gain by incorporating the benefits of both systems. Typically, in parallel applications, the number of transactions can vary anywhere from a single transaction to a large number of transactions. It is important to note that not all transactions are identical. Each transaction has its own characteristics in terms of transaction size, read-set size and write-set size. Depending on these characteristics of a transaction, either HTM or STM can be a better choice for implementation. We exploit the decision tree [9] to predict whether HTM or STM is faster for a given transaction. The decision tree receives

978-1-4673-8776-7/16 $31.00 © 2016 IEEE
DOI 10.1109/PDP.2016.85

input parameters (such as transaction size, transactional write ratio, etc.) and predicts the optimum TM system for a transaction. Then, a programmer or a compiler modifies the source code of the application based on predictions made by the decision tree. Our adaptive system supports both HTM and STM with the aim of reducing execution time of transactions with different characteristics. In this work, we use RTM [5] for hardware transactions and TinySTM [2] for software transactions. It is important to note that while we use RTM and TinySTM in this work, our adaptive system is general, and can be implemented using any HTM or STM system.

In summary, we make the following contributions:

- We show that there is no single TM system that works well across all applications. Depending on applications' characteristics, one system might be better than the other.
- We propose an adaptive system, which predicts the optimum TM system for a given transaction, statically. The adaptive system relies on the prediction of the decision tree to select either HTM or STM.
- Our evaluations using STAMP [10], NAS [11], and DiscoPoP [12] benchmark suites reveal that on average, the adaptive system is able to improve speed of transactional applications by 20.82%.

The rest of the paper is organized as follows. Section II provides background information on Intel's RTM [5], TinySTM [2], and the decision tree prediction model [9] and also motivation behind this work. Section III describes details of the adaptive system. Section IV analyzes the results of the experimental evaluations. Section V presents the related work. Finally, section VI concludes the paper.

II. Background and Motivation

A. Intel's Restricted Transactional Memory

Along with the release of the Haswell processor, Intel introduced Transaction Synchronization Extensions (TSX) to provide programming support for hardware transactional memory. Intel's TSX has two variants: Hardware Lock Elision (HLE) and Restricted Transaction Memory (RTM). HLE is a legacy compatible instruction set that uses lock acquisitions strictly in hardware. In this paper, the main focus is on RTM.

The programming model of Intel's RTM allows programmers to mark regions of a code to be executed transactionally. In order to access RTM's functionality, there are 3 new instructions: XBEGIN, XEND and XABORT. A transaction is initiated with the instruction XBEGIN. Inside of the transaction, RTM uses the processor's cache to track read and write sets of transactions. These read and write sets are monitored in the granularity of cache blocks. To state the end of a transaction, the instruction XEND is used. The XEND instruction commits any changes done to the shared memory. To explicitly abort a transaction, XABORT is used inside of the transactional code.

In RTM, the conflict detection is handled through the cache coherency protocol. If two transactions access a shared memory location and if at least one of them writes into the location, the cache coherency protocol detects the conflict. In the event of conflict, only one transaction can proceed while the rest should abort. RTM follows the eager policy to resolve conflicts. In eager policy, as soon as a transactional write operation results in conflict, RTM will then abort the conflicting transactions and allows only one transaction proceed. Eager policy improves utilization of processor resources as a conflicting transaction is aborted immediately and is not postponed to the commit time.

There are numerous reasons for transactional aborts. The primary causes of a transactional abort in RTM are conflicts over shared memory locations and limited capacity in hardware resources. For example, the size of data accessed in a transaction may exceed cache capacity that triggers a transactional abort. Other aborts in RTM are caused by software/system operations such as page faults, context switching, I/O operations, etc. In RTM, transactional aborts are identified in the EAX register [5]. The EAX register carries an 8-bit code that specifies the cause of the transactional abort. When a transaction is aborted, all the changes made to the memory are discarded and an abort code is sent from the EAX register, stating the cause of the abort. RTM does not guarantee forward progress which means a transaction may not eventually commit if it solely relies on RTM. Thus, a transaction requires software based fallback policy to provide forward progress guarantees.

B. TinySTM

In this paper, we incorporate an already established state-of-the-art STM system, called TinySTM [3]. This system is a word-based STM implementation that uses conventional lock-based mechanisms to protect shared memory locations. It also uses a time-based design to guarantee that all transactions access consistent memory states. TinySTM uses encounter-time locking which is beneficial for detecting conflicts earlier (increasing transaction throughput). When compared to commit-time locking, TinySTM improves utilization of processor resources as a doomed transaction is aborted immediately. Also, encounter-time locking efficiently manages the read and write operations without requiring complex mechanisms. TinySTM was chosen as the STM counterpart for the adaptive system due to having better performance compared to other STM systems (such as TL2 [1]).

C. Motivation

Both STM and RTM have benefits and limitations that either improves or penalizes performance in certain applications. One of the most important differences between RTM and STM is transactional overhead. In RTM, the processor is responsible for transactional execution and conflict detection, which incurs much less overhead and exhibits better overall performance. On the other side, in STM, there is extra overhead for software based conflict detection and data versioning (such as initiating a transaction, validating transactional data, transactional

commits, etc. [3]). This greatly hampers the overall performance in STM systems. Another important aspect of the two systems is flexibility. In RTM, the processor oversees all memory accesses, which in-hand provides strong isolation but relies solely on hardware resources. This results in complexity issues (fallback policy is needed) that lead to a higher probability of transactional aborts and in certain cases a performance slowdown when compared to STM. On the other hand, STM delivers a flexible system in which there is no resource limitation and the underlying system deals with majority of the complex synchronization issues, leading to less transactional aborts and a better overall performance in some applications when compared to RTM.

Figure 1 represents a normalized comparison graph between RTM and STM using 12 benchmarks taken from STAMP [10], NAS [11], and DiscoPop [12] benchmark suites. In each benchmark, the number of threads varies between two and eight (for detail of experimental framework, refer to Section IV). In Figure 1, measurement reading less than one favors RTM, while greater than one favors TinySTM. There is a vast discrepancy between both systems, primarily due to the transaction characteristics within a given benchmark such as transaction size, write-set size and read-set size. In small benchmarks where working set of the benchmark fits in the L1 cache, i.e. Montecarlo, RTM outperforms TinySTM. On the other hand, TinySTM outperforms RTM in benchmarks consisting of larger transaction sizes, i.e. Labyrinth. The number of transactions within a benchmark varies and the characteristics from one transaction to another transaction also vary. By introducing our adaptive system, we will be able to switch between RTM and TinySTM within a benchmark and achieve better performance.

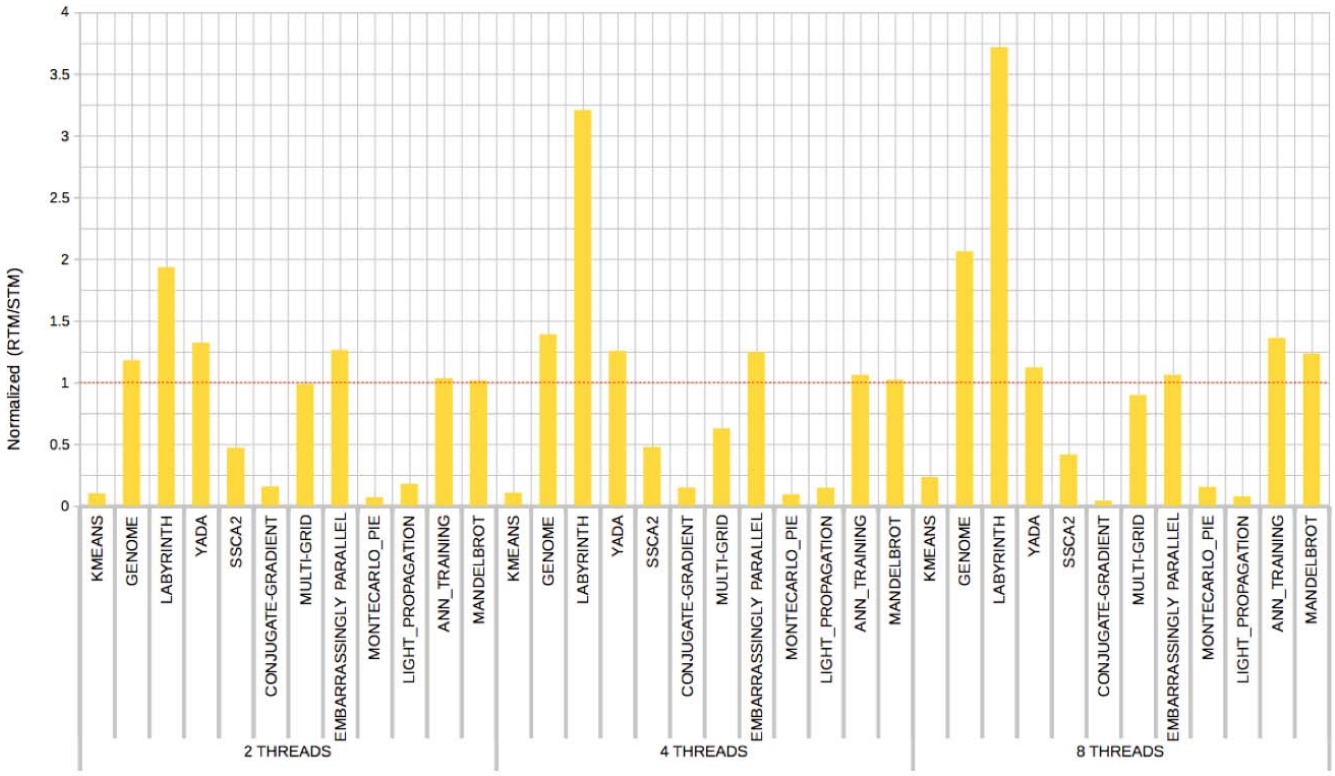

Figure 1. Normalized Transactional Execution time of RTM relative to TinySTM.

III. Adaptive System Design

In this section, we describe our proposed Adaptive system which supports both hardware and software transactions within an application, while guaranteeing forward progress. Our adaptive system uses Haswell's RTM [5] and a modified version of TinySTM [2]. Both of these systems work in conjunction with each other in order to achieve dynamic switching. One of the features of the adaptive system is that it switches between HTM and STM in transaction granularity. In parallel computing, the term granularity is defined as the amount of real work done in a parallel task. With transaction granularity, the focus is on the basis of individual transactions rather than an entire application. This fine-grained granularity system increases performance gains, while a coarse-grained granularity system misses many opportunities for speedup. However, to avoid overhead, adaptive system does not execute HTM and STM simultaneously. Simultaneous execution of HTM and STM requires communication between in-flight hardware and software transactions. A metadata should record transactional data and each transaction should check the metadata when it accesses a transactional variable. Doing so significantly increases execution time and hurts performance,

especially in applications with low conflict rate. To avoid this performance penalty, we allow a transaction executes either in hardware or software, but not both.

The implementation of RTM programs was based on the programmability references from Intel's TSX manual [5]. The key factors in the RTM programs is retry count, fallback policy, and abort status. The retry count is the number of times an aborted transaction retries execution. This is important because in RTM, transactions have an abundant reasons to abort (refer to section II. A). By retrying an aborted transaction 'x' number of times, there is a possibility that the transaction can eventually commit in hardware. Once the retry threshold is reached, the fallback policy is applied. The fallback policy that is used is a global lock mechanism. In our adaptive system, the retry count is set to 4. Based on our experimental simulations, the retry count of 4 is the best option that produces optimal performance. It is possible to have a higher retry count, but it can hurt performance as retrying a transaction that aborts over and over increases execution time. Also, having a low retry count can cause the fallback policy to be executed too early. RTM's EAX status register was utilized to track the types of transactional aborts.

A. Synchronization of RTM and STM

This section explains how RTM and STM are synchronized. We need to guarantee that in-flight hardware and software transactions do not execute simultaneously. This is very crucial because if there are any issues it can stall an application from executing correctly or crash entirely. It can also lead to incorrect updates to the shared variables by either one of the systems. To enable mutual-exclusion of RTM and STM, we exploit conditional variable. The pseudo code in Figures 2 and 3 show how synchronization is handled between the two systems.

```
1:  tx_start(int rtm_n_stm)
2:  {
3:    ...
4:   if(rtm_n_stm == 1)
5:    {
6:      pthread_mutex_lock(&rtm_stm_sync_mutex);
7:      while (num_in_flight_stm > 0)
8:              pthread_cond_wait(&sync_cond_rtm, &rtm_stm_sync_mutex);
9:      num_in_flight_rtm++;
10:
11:     pthread_mutex_unlock(&rtm_stm_sync_mutex);
12:   }
13:
14:  if(rtm_n_stm == 0)
15:   {
16:     pthread_mutex_lock(&rtm_stm_sync_mutex);
17:     while (num_in_flight_rtm > 0)
18:             pthread_cond_wait(&sync_cond_stm, &rtm_stm_sync_mutex);
19:     num_in_flight_stm++;
20:
21:     pthread_mutex_unlock(&rtm_stm_sync_mutex);
22:   }
23:  ...
24: }
```

Figure 2. Pseudo code for synchronization of RTM and STM in tx_start().

The synchronization occurs inside the functions tx_start() and tx_commit() which depict the start and end of a transaction, respectively. These functions have other code sequences but are taken out in order to only focus on the synchronization part. The input arguments of the two functions show whether the corresponding transaction is executed in hardware or software. A hardware transaction first checks if there is any in-flight software transaction (line 7). If a software transaction is executing, then the hardware transaction waits (line 8). Then, the hardware transaction increments num_in_flight_rtm which is a counter and shows the number of in-flight hardware transactions (line 9). A global lock (rtm_stm_sync_mutex) is used to guarantee atomicity of accesses to the shared variables in tx_start() and tx_commit(). It is important to note that the overhead of the global lock is very low as it is held by transactions for a short period of time. The code for software transactions (lines 14-22) is similar. When a hardware transaction commits, (lines 28-35), it decrements num_in_flight_rtm counter (line 31). If the counter is zero, then it broadcasts a signal to all software transactions waiting for in-flight hardware transactions to finish (line 33). The same procedure is followed for software transactions (lines 37-44).

```
25: tx_commit(int rtm_n_stm)
26: {
27:   ...
28:  if(rtm_n_stm == 1)
29:    {
30:      pthread_mutex_lock(&rtm_stm_sync_mutex);
31:      num_in_flight_rtm--;
32:      if(num_in_flight_rtm == 0)
33:            pthread_cond_broadcast(&sync_cond_stm);
34:      pthread_mutex_unlock(&rtm_stm_sync_mutex);
35:    }
36:
37:  if(rtm_n_stm == 0)
38:    {
39:      pthread_mutex_lock(&rtm_stm_sync_mutex);
40:      num_in_flight_stm--;
41:      if(num_in_flight_stm == 0)
42:            pthread_cond_broadcast(&sync_cond_rtm);
43:      pthread_mutex_unlock(&rtm_stm_sync_mutex);
44:    }
45:  ...
46: }
```

Figure 3. Pseudo code for synchronization of RTM and STM in tx_commit().

B. Implementation of Decision Tree Prediction Module

Decision tree is an effective method of supervised machine learning that exhibits an accurate prediction based on a group of datasets [9]. Our Adaptive system exploits a decision tree prediction module (C4.5 algorithm [4]) to be able to predict which TM system is the better choice for a given transaction. The basic functionality of C4.5 is to build a tree from a set of training datasets and the resulting tree is used to predict the optimum TM system. This process can be broken down into two phases: training phase and testing phase.

1) Training Phase

The training phase is conducted to attain a prediction model based on decision tree. The input datasets are based on the following transaction parameters: transaction size, read-set size, write-set size, and write-ratio (ratio of shared writes and total number of shared accesses in a transaction).

The output of the decision tree is a binary bit that indicates whether RTM or STM is better for a given transaction. These parameters are important in terms of the behaviors of both RTM and TinySTM. In RTM, it favors small sized transactions as well as small working set size (number of distinct memory locations accessed). While in STM, there is much more flexibility and offers better performance than RTM for large transaction size and large working set size. The training phase consists of a set of benchmarks that are chosen based on small, medium and large transaction sizes and working set sizes from all the 3 benchmark suites (STAMP, NAS and DiscoPoP). The following are the benchmarks used for the training phase: GENOME, LABYRINTH, YADA, Embarrassingly Parallel, Montercarlo_Pie and Light_Propogation. Benchmarks are executed twice: once using RTM and the other time using TinySTM. Decision tree is trained based on statistics generated by RTM and TinySTM. This procedure was done separately for 2, 4, and 8 number of threads because the characteristics of a transaction can vary as the thread count increases.

Table I shows an example of benchmark YADA and the parameters associated with its transactions. These parameters were used for training due to specific behaviors of each system. Benchmark YADA contains 5 transactions in which each transaction has its own unique set of characteristics.

One way to measure transaction size (the 6th column in Table I) is to count the number of C code lines in transactions. However, execution time of C programs changes from one line to the other by a large margin. We need a fine granularity metric for transaction size. Since all C codes are compiled to assembly instructions, we use number of assembly instructions to measure transaction size

In large transactions, STM performs better than RTM primarily due to capacity overload of hardware resources. Another critical behavior of a transaction is working set size (read/write accesses). RTM performs well for transactions that consist of low to medium working set size, while STM performs well for large working set size. This is due to the hardware constraints associated with RTM which caps the threshold for performance gain in transactions with large working set sizes. In YADA, there are 4 transactions with a transaction size that ranges from 95-115. For these transactions, RTM executes faster than STM. The remaining transaction has a size of 626 and contains a very large working set size in which STM greatly outperforms RTM. By training the decision tree using all parameters of the training benchmarks, it is possible to achieve accurate predictions.

TABLE I. CHARACTERISTICS OF BENCHMARK YADA CONSISTING OF FIVE TRANSACTIONS

TX #	STM Time(ms)	RTM Time(ms)	Read-set Size	Write-set Size	TX Size	Write Ratio
TX1	291	113	2525298	1219387	101	0.3256
TX2	523	48	580197	0	115	0
TX3	39833	51061	10396152	24145158	626	0.1884
TX4	52	24	0	464996	95	1
TX5	144	66	1127133	505601	109	0.3096

2) Testing Phase

The testing phase is conducted to predict whether RTM or STM is better for a given transaction. This testing phase consists of 6 different benchmarks, which are: Conjugate-Gradient, Multi-Grid, KMEANS, SSCA2, Ann_Training and Mandelbrot. The C4.5 algorithm of the decision tree applies pruning to increase the accuracy of the prediction. Pruning is the basis of increasing the accuracy of unseen data. The decision tree is designed to give an accurate prediction, which means that there is no guarantee that the prediction is correct all the time. This is due to the parameters that impact execution time of transactions. These parameters vary from one benchmark to another. Table II is an example of the prediction of the decision tree for benchmark CG (Conjugate-Gradient). D. T prediction in the table stands for decision tree prediction. The decision tree prediction is based on the dataset of the training phase. The optimum system represents the system that executes the fastest.

TABLE II. CONJUGATE-GRADIENT BENCHMARK COMPARING DECISION TREE PREDICTION WITH OPTIMUM SYSTEM

TX #	STM Time(ms)	RTM Time(ms)	D.T prediction	Optimum prediction
TX1	4	21	RTM	STM
TX2	83391	9664	RTM	RTM
TX3	97	809	STM	STM
TX4	14	2	STM	RTM
TX5	4	20	RTM	STM
TX6	172	1873	STM	STM

This table indicates that the decision tree predicted the best system at a rate of 50% (3/6 transactions). Even though 50% accuracy seems poor, it is actually very accurate in terms of transaction execution time greater than 100ms. Approximately, 3 out of the 6 transactions have an execution time greater than 100ms (for both RTM and STM), in which the decision tree accurately predicted the correct system to use. The miss-predictions for the transactions with an execution time less than 100ms are not important as small transactions have insignificant impact on performance. Our adaptive system works alongside the predictions resulted by the decision tree. Based on the prediction, either a programmer or a compiler will statically change the source code for the adaptive system. The adaptive system will then run the benchmark, which consists of both hardware and software transactions to achieve a performance gain.

IV. EXPERIMENTAL RESULTS

For our adaptive system, it is important to simulate both STM and RTM on the same commodity processor. The experimental setup consisted of 4th generation Intel Core i7 processor comprising of four physical cores that can run up to eight threads simultaneously (hyper-threading). Each core consists of two 8-way 32KB L1 cache, 256 KB L2 cache, and 8 MB of L3 cache. We compile all benchmarks using gcc 4.8.1. We use the –mrtm flag to access the Intel's TSX intrinsic. For evaluation, the benchmarks from the testing phase are used. This includes benchmarks Conjugate-Gradient, Multi-Grid, KMEANS, SSCA2, Ann_Training and Mandelbrot. We did not include the training benchmarks for

our evaluation because we wanted to have discrete analysis based on the decision tree prediction. Therefore, the focus was on attaining a prediction based on the training benchmarks then applying the prediction to another set of benchmarks (testing benchmarks). Figure 4 represents normalized speedup comparison between the adaptive system and TinySTM. A benchmark that consists of a value less than 1 shows speed-up for the adaptive system. The benchmarks Conjugate-Gradient, Kmeans, and SSCA2 have a significant speedup over STM. The rest of the benchmarks, Multi-Grid, Ann_Training and Mandelbrot have a normalized speedup value of 1, which indicates that the prediction used for the adaptive system heavily favored TinySTM. On average, speed-up is 34.31%, 34.44%, and 34.35% for 2, 4 and 8 threads, respectively.

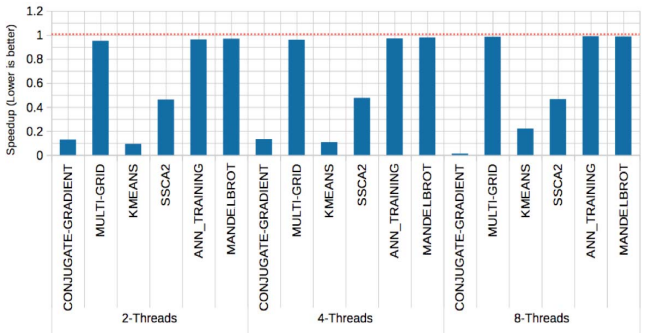

Figure 4. Normalized Speedup comparison between adaptive system and TinySTM.

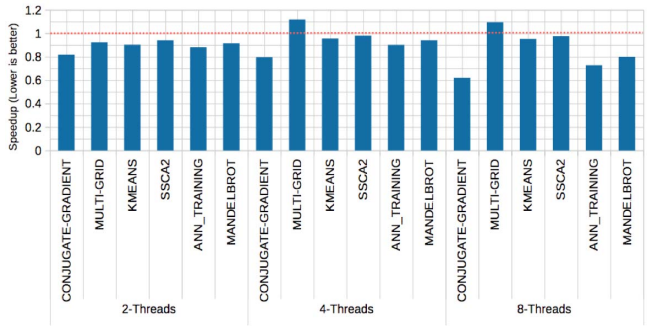

Figure 5. Normalized Speedup comparison between sdaptive system and RTM.

Figure 5 represents Normalized Speedup comparison between the Adaptive system and RTM. On average, speedup is 5.88%, 5.16% and 11.79% for 2, 4 and 8 threads, respectively. The benchmarks that have a normalized speedup less than one, indicate that the adaptive system achieves speedup. At 4 and 8 threads, benchmark Multi-Grid indicates a slowdown when compared to the baseline RTM. This is due to the decision tree prediction that incorrectly predicted the wrong system to execute for that specific benchmark. Table III shows transaction parameters of Multi-Grid when the number of threads is 4. Multi-Grid benchmark consists of two transactions in which the decision tree predicts correctly for only one of the two transactions. The other transaction (TX2) is incorrectly predicted and this results in slowdown of the adaptive system compared to the baseline RTM. There are a few reasons why RTM executes better than STM even though the transaction and working set sizes are very large. The primary reason is the abort ratio of this benchmark. In RTM, capacity induced aborts dramatically hamper the performance of transactional executions. Yet, for benchmark Multi-Grid there is a total abort ratio of 11.46% and out of that, only 9.54% consists of capacity aborts. This means that there is a low abort rate as this benchmark has a higher percentage of successfully committing transactions. Also, since the capacity abort rate is very low, this benchmark executes efficiently in RTM thus achieving a better performance.

TABLE III. TRANSACTION PARAMETERS AND EXECUTION TIME FOR MULTI-GRID BENCHMARK WHEN NUMBER OF THREADS IS 4.

TX #	STM Time(ms)	RTM Time(ms)	Read-set Size	Write-set Size	TX Size	Write Ratio	D. T. Pred.	Opt. Sys.
TX1	120	60	64	64	130	0.5	RTM	RTM
TX2	18818	16990	8008	8008	276	0.5	STM	RTM

A. *Energy Expenditure Analysis*

An important aspect of a computational platform is energy efficiency. With modern technology (laptops, cell phones, tablets, ext.) relying heavily on battery power, it is essential to expend an efficient amount of energy as possible. We used Intel's runtime average power limit monitor (RAPL) to measure energy expenditure of the benchmarks [13]. RAPL relies on a set of hardware counters inside the processor which provides energy and power consumption information.

The energy readings are measured on the entire application for our adaptive system and then compared the statistics to the baseline RTM and TinySTM. First, energy measurements are taken for each system and an analysis is made. RTM is more energy efficient than TinySTM as RTM exploits hardware resources and does not incur the software overhead of TinySTM. By implementing our adaptive system, there is a possibility that by switching to RTM (when possible), it may be more energy efficient than STM. Furthermore, the adaptive system will also incorporate STM, meaning the energy efficiency readings compared to RTM does not result in efficiency. To take into account the impact of both energy and performance, we use energy-delay to compare adaptive system with RTM and TinySTM.

Figure 6 depicts normalized energy-delay of our adaptive system compared to TinySTM. For this evaluation, only the benchmarks in the testing phase are used in order to have a realistic evaluation based on the decision tree predictions. Since this is a normalized graph, values less than 1 depict energy efficiency for the adaptive system. In all the testing benchmarks, our adaptive system is 42.11% more energy efficient than TinySTM. This is because for certain benchmarks that consist of low-medium transaction and working set sizes, by implementing these transactions in RTM, we are able to save energy. If all the transactions are implemented in STM, then there will be additional overhead for each transaction initiated. Figure 7 depicts normalized

energy-delay graph comparison between RTM and our Adaptive system. Our adaptive system is not energy efficient when compared to RTM. This is simply due to the overhead associated with switching into STM. There is extra overhead, when initiating and overseeing a transaction in STM, which expends extra energy. Thus, since our adaptive system incorporates both systems, the energy efficiency drops when compared to RTM.

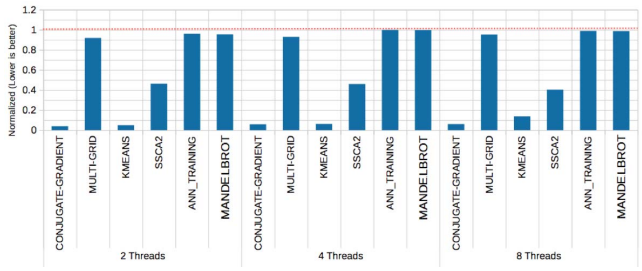

Figure 6. Normalized Energy-delay comparison between adaptive system andTinySTM.

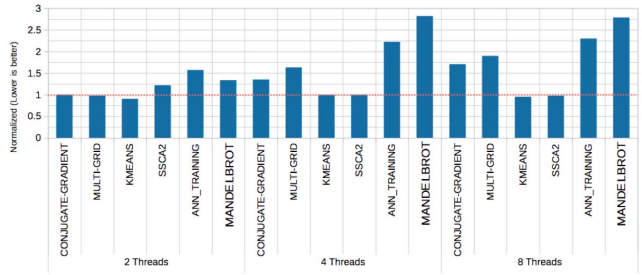

Figure 7. Normalized Energy-delay comparison between adaptive system and RTM.

V. RELATED WORK

Research in Transactional Memory has progressed positively in the past decade with majority of studies focused on STM systems. Recently, Intel [5] and IBM [7, 8] released chip multiprocessors that include hardware support for transactional memory. This has sparked a new interest on successfully coupling the new hardware support with software approaches. An intensive evaluation of Intel's TSX was presented by Yoo et al. [14], in which they used STAMP benchmarks to compare RTM and TL2. They investigated significant performance differences between both systems as well as identifying potential pitfalls with hardware resources that can lead to performance loss. A continuation of this study was presented by Wang et al. [15] where they focused on the relationships between transaction size, write-ratio, retry count and abort ratio. They conducted these tests using micro-benchmarks and compared performance to lock based mechanisms. Our work is different as we offer a design technique that incorporates both RTM and TinySTM to boost performance of TM applications.

Calciu et al. [16] presented Invyswell, a hybrid transactional memory system that incorporates RTM and InvalSTM. They investigate RTM's limitations and provide InvalSTM as a fallback policy instead of using lock mechanisms. For Invyswell, each transaction is first tried in hardware. If the hardware abort status (EAX register) suggests that a transaction is unlikely to succeed in hardware, then it is retried in InvalSTM. They also incorporate fail-fast optimization technique. This technique is used for an application with high contention, which results in a higher probability of hardware resources reaching capacity limit. It is used to identify certain cases when RTM is wasting work with too many retries which eventually calls the fallback policy once the retry threshold has been met. In this study, energy expenditure was not included and this is primarily due to being a dynamic approach (runtime overhead is high) as well as having InvalSTM as a fallback policy, which incurs extra overhead. In our study, the adaptive system is static and its runtime overhead is low. Also, we do not use STM as a fallback policy for RTM. Instead we implement independent switching between RTM (lock mechanism for fallback policy) and STM. Energy delay measurements shows that our adaptive system is much more efficient than baseline STM.

Pereira et al. [17] presented an extensive evaluation of Haswell's Transactional Memory performance. They focus on RTM's forward-progress polices since Intel's TSX does not guarantee that a transactional execution will commit. This technique is to retry the execution of a transaction with or without a time delay to attempt to complete the transaction execution speculatively. They introduced an optimized policy called SerControl where the focus is on the type of transactional abort in RTM by using the EAX register status bit. If the transaction is aborted due to conflict or capacity consecutively, SerControl will serialize the transaction by using a lock. If the cause of abort is not conflict or capacity, then the Max retry policy is applied. In our study we incorporated the concepts of forward progress policies and applied it to the RTM system. Although, the notion of having an efficient forward progress policy is important, the actual performance gains are negligible. Pereira et al. [17] do not show the comparisons between the proposed RTM forward progress policy and another STM system. On the other side, we investigated the behaviors of a transaction that best suit each TM system. If a transaction consists of a very large transaction size as well as a very large working set size, having an optimized forward progress policy will not change the fact that RTM will perform poorly. In this case, our adaptive system will automatically execute the optimal system based on the characteristics of a transaction.

Castro et al. [18] presented a dynamic approach to do efficient thread mapping using machine learning which relies on matching the behavior of an application with the system characteristics. The basis of thread mapping assigns threads dynamically to the processing cores in order to reduce the latency associated with memory hierarchy. This is accomplished by monitoring the status of a transaction as well as the STM system at specific intervals. Following each interval, the thread mapping strategy is applied based on the decision tree prediction model using ID3 algorithm [19]. In our study, we incorporated the decision tree to

predict the optimum system for a given transaction. This paper proves that by incorporating a decision tree, we are able to classify a transaction's parameters in order to predict the optimum system that achieves the best performance. The decision tree algorithm used in the paper is ID3 while our study focused on the C4.5 algorithm. C4.5 is an enhanced version of ID3, as it also supports continuous attributes that result in better performance. This paper also follows a procedure of attaining a training set of benchmarks and a testing set of benchmarks. By separating the training and testing, it is possible to achieve results based on the prediction of the decision tree itself. For our study, a training set of benchmarks consisted in the basis of low, medium, large transaction sizes as well as low, medium, large working set sizes.

VI. CONCLUSION

In this paper, we proposed an adaptive system that exploits both STM and HTM at transaction granularity. We developed a synchronization technique to seamlessly switch between RTM and TinySTM based on the characteristics of a transaction. We exploit the decision tree to predict the optimum system for each transaction in a given application. The decision tree is a form of supervised machine learning to classify the input transaction parameters (such as transaction size, transactional write ratio, etc.). This leads to an accurate prediction to execute the optimum TM system. The evaluation consisted of three parallel benchmark suites separated into the training phase and the testing phase. The decision tree attains all transactional parameters from the benchmarks in the training phase and predictions are created for varying number of threads. These predictions are then evaluated on the testing phase which reveals that the adaptive system is able to improve transactional execution time and energy-delay.

ACKNOWLEDGMENT

This work was supported by the Natural Sciences and Engineering Research Council of Canada.

REFERENCES

[1] David Dice, Ori Shalev, and Nir Shavit, Transactional Locking II, In Proceedings of the 20th International Symposium on Distributed Computing , pages 194-208, September 2006.

[2] P. Felber, C. Fetzer, and T. Riegel. Dynamic Performance Tuning of Word-Based Software Transactional Memory. In Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Salt Lake City, UT, Feb. 2008.

[3] Cunha, G., J. Lourenço, and R. J. Dias, "Consistent State Software Transactional Memory", IV Jornadas de Engenharia de Electrónica e Telecomunicações e de Computadores (JETC'08), Lisboa, Portugal, ISEL - Instituto Superior de Engenharia de Lisboa, pp. 251–256, 2008.

[4] J. R. Quinlan, C4.5: Programs for Machine Learning", Morgan Kaufmann Publishers, ISBN- 0080500587, 1993.

[5] Intel Corporation, "Chapter 12: Intel's Transactional Synchronization Extensions (TSX)," Jul. 2013. Website link: http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-optimization-manual.html

[6] D.E. Culler, J. Singh, A. Gupta, Parallel Computer Architecture: A Hardware/Software Approach, Morgan Kaufman Publishers, San Francisco, CA, 1999.

[7] A. Wang, M. Gaudet, P. Wu, J. N. Amaral, M. Ohmacht, C. Barton, R. Silvera, and M. Michael. Evaluation of blue gene/q hardware support for transactional memories. In Proceedings of the 21st international conference on Parallel architectures and compilation techniques, PACT '12, pages 127–136, New York, NY, USA, 2012. ACM.

[8] C. Jacobi, T. Slegel, and D. Greiner. Transactional memory architecture and implementation for ibm system z. International Symposium on Microarchitecture, MICRO '12, pages 25–36, Washington, DC, USA, 2012. IEEE Computer Society.

[9] Quinlan, J. R. Induction of Decision Tree. Machine learning, 1986, 1(1): 81-106.

[10] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun, "STAMP: Stanford Transactional Applications for Multi-Processing," in *The IEEE International Symposium on Workload Characterization (IISWC)*, Seat- tle, WA, USA, Sep. 2008, pp. 35–46.

[11] D. Bailey, E.Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T.Lasinski, R. Schreiber, H. Simon, V.Venkatakrishnan and S. Weeratunga. The NAS parallel Benchmarks. RNR Technical Report RNR-94-007, March 1994.

[12] Zhen Li, Ali Jannesari, Felix Wolf. Discovery of Potential Parallelism in Sequential Programs. In Proceedings of the 42nd International Conference on Parallel Processing Workshops (ICPPW), Workshop on Parallel Software Tools and Tool Infrastructures (PSTI), Lyon, France, pages 1004-1013, October 2013.

[13] Intel Architecture Software Developer's Manual: System Programming Guide, June. 2013.

[14] R. M. Yoo, C. J. Hughes, K. Lai, and R. Rajwar, "Performance evaluation of intel® transactional synchronization extensions for high-performance computing," in Proceedings of the International Con- ference on High Performance Computing, Networking, Storage and Analysis, ser. SC '13, New York, NY, USA, 2013, pp. 19:1–19:11.

[15] M. D. Wang, M. Burcea, L. Li, S. Sharifymoghaddam, G. Steffan, and C. Amza, "Exploring the performance and programmability design space of hardware transactional memory," in The ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Trans- actional Computing (TRANSACT), Raleigh, NC, USA, Mar. 2014.

[16] Irina Calciu , Justin Gottschlich , Tatiana Shpeisman , Gilles Pokam , Maurice Herlihy, Invyswell: a hybrid transactional memory for haswell's restricted transactional memory, Proceedings of the 23rd international conference on Parallel architectures and compilation, August 24-27, 2014.

[17] Marcio Machado Pereira, Matthew Gaudet, José Nelson Amaral, and Guido Araújo. 2014. Multi-dimensional Evaluation of Haswell's Transactional Memory Performance. In Proceedings of the 2014 IEEE 26th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD '14).

[18] Márcio B. Castro, Luís F. Góes, Luiz Gustavo Fernandes, Jean-François Méhaut: Dynamic Thread Mapping Based on Machine Learning for Transactional Memory Applications. Euro-Par 2012: 465-476

[19] Quinlan, J. R. C4.5: Programs for Machine Learning. Morgan Kaufmann Publishers, 1993.