


## RESEARCH ARTICLE

# Improving performance of transactional memory through machine learning

Yang Xiao<sup>1</sup> | Thiresan Jeyakumaran<sup>1</sup> | Ehsan Atoofian<sup>1</sup>  | Ali Jannesari<sup>2</sup><sup>1</sup>Electrical Engineering Department, Lakehead University, Thunder Bay, ON, Canada<sup>2</sup>University of California, Berkeley, CA, USA**Correspondence**Ehsan Atoofian, Electrical Engineering Department, Lakehead University, Thunder Bay, ON, Canada.  
Email: atoofian@lakeheadu.ca**Summary**

Transactional memory (TM) is a programming paradigm that facilitates parallel programming for multi-core processors. In the last few years, some chip manufacturers provided hardware support for TM to reduce runtime overhead of Software Transactional Memory (STM). In this work, we offer two optimization techniques for TMs. The first technique focuses on Restricted Transactional Memory (RTM) in Intel's Haswell processor and shows that while in some applications, RTM improves performance over STM, in some others, it falls behind STM. We exploit this variability and propose an adaptive technique that switches between RTM and STM, statically. The second technique focuses on the overhead of TM and enhances the speed of the adaptive system. In particular, we focus on the size of transactions and improve performance by changing the transaction size. Optimizing the transaction size manually is a time-consuming process and requires significant software engineering effort. We use a combination of Linear Regression (LR) and decision tree to decide on the transaction size, automatically. We evaluate our optimization techniques using a set of benchmarks from NAS, DiscoPoP, and STAMP benchmark suites. Our experimental results reveal that our optimization techniques are able to improve the performance of TM programs by 9% and energy-delay by 15%, on average.

**KEYWORDS**

decision tree, energy, linear regression, performance, restricted transactional memory, software transactional memory

## 1 | INTRODUCTION

Transactional Memory (TM) is emerging as a popular technology to address the difficulty of parallel programming for multi-core processors. In TM, a programmer simply specifies the code regions operating on shared data that should execute atomically with respect to all other transactions. The underlying system guarantees the atomicity of accesses to the shared data as well as correctness of transactional execution. This is in contrast to conventional lock-based parallel programming, which requires significant programming effort to avoid synchronization bugs such as deadlock, priority inversion, lock convoying, etc.<sup>1</sup> There are two major types of TM: Software Transactional Memory (STM) and Hardware Transactional Memory (HTM).

STM uses software support to track data written and read by transactions and to detect conflict between concurrent transactions. There have been significant efforts to implement efficient STM, and this led to the development of a variety of STM systems such as TL2,<sup>2</sup> TinySTM,<sup>3</sup> and McRT-STM.<sup>4</sup> While STM is flexible and runs on existing hardware, it results in runtime overhead due to transactional bookkeeping. For each memory location accessed by a transaction, STM requires a metadata to track the record of accesses to the location. When the transaction writes into the location, it checks all transactions that already accessed the location to detect conflicts. This may lead to poor performance and may slow down STM compared to a serial program.

To cope with the overhead of STM, some chip manufacturers such as Intel<sup>5</sup> and IBM<sup>6,7</sup> offered processors with hardware support for TM. In this work, we exploit Restricted Transactional Memory (RTM)<sup>5</sup> in Intel's Haswell processor as we do not have access to processors of other vendors. However, we believe that our techniques can be applied to other processors as HTMs of different vendors are similar. In RTM, data caches are used to track data read and written by transactions. The cache coherence protocol is changed to detect conflicts between concurrent transactions.<sup>8,9</sup> This

removes the burden of transactional bookkeeping from software and reduces runtime overhead. However, RTM has its own set of disadvantages and constraints. RTM uses a best-effort policy for the execution of transactions, which means that it does not guarantee that a transaction eventually commits in hardware successfully. This happens under different conditions. For example, if the working set of a transaction does not fit in the L1 data cache, then the transaction aborts. It is up to a programmer to provide an alternative approach for the execution of the transaction, called the fall-back path. The fall-back path can be as simple as a lock. There are other conditions that lead to a transaction being aborted such as context switching, I/O interrupt, etc. Thus, both STM and HTM have their own set of advantages and disadvantages.

In this work, we offer two techniques to enhance the performance of TMs.

The first scheme is an adaptive system that dynamically switches between STM and HTM. The goal is to incorporate a mechanism that benefits from both TM implementations and achieves performance gains of the better of the two. Each transaction has its own set of characteristics that makes one of the two TMs a better choice for execution. For example, if a transaction has a large read-set, then the transaction aborts if it executes in hardware. The main reason is that hardware buffers overflow as there are too many entries in the read-set. However, STM is not constrained by read-set size. On the other side, if the read-set of a transaction is small, HTM is a better option as it does not incur overhead of STM. To decide whether HTM or STM should be used for a given transaction, we exploit decision tree.<sup>10</sup> Through profiling, the decision tree receives an input vector composed of transactional parameters such as transaction size, read-set size, etc., and predicts whether the transaction should be executed in HTM or STM. Then, a programmer or a compiler should change the transaction code based on the prediction made by the decision tree. In this work, we use TinySTM<sup>3</sup> as an implementation of STM. It is important to note that although we use RTM and TinySTM to evaluate our proposed techniques, our techniques are general and are independent from the choice of the underlying HTM or STM system.

The second scheme, called adaptive+, improves performance over adaptive by optimizing the transaction size. Generally speaking, the size of a transaction is an important factor that impacts the execution time of the transaction. If a transaction is small, then the overhead of TM may exceed its performance gain. This may limit speedup and even make the TM slower than its sequential version. On the other side, if a transaction is large, then the cost of abort might be too high, offsetting the performance gain of TM. To decide on the transaction size, we need a mechanism that automatically determines the near-optimum transaction size. Clearly, a try-and-error approach is not an option as it is not feasible for use in an application with a large number of transactions. We use a combination of Linear Regression (LR)<sup>11</sup> and decision tree<sup>10</sup> to decide on the transaction size automatically. Through profiling, LR receives parameters of a non-optimized transaction such as write-set size, size of next transaction, etc., and tries to predict the near-optimum transaction size. Decision tree enhances the accuracy of predictions by classifying transactions into multiple categories and then uses a different LR model for each category.

In summary, we make the following contributions.

- We show that neither HTM nor STM is optimum across applications. Even within an application, a program goes through different phases and the optimum TM may vary across the phases.
- We propose an adaptive system that exploits the better of the two TM systems and switches between the two. We decide on TM implementations statically and using a decision tree.
- We show that transaction size plays an important role in the performance of TM applications. We propose adaptive+, which enhances the adaptive system by changing the transaction size statically and before the runtime. We exploit a combination of LR and decision tree to decide on the transaction size.
- We evaluate our proposed schemes using NAS,<sup>12</sup> DiscoPoP,<sup>13</sup> and STAMP<sup>14</sup> benchmark suites. We show that adaptive+ improves both speed and energy over adaptive, HTM, and STM systems.

The rest of the paper is organized as follows. In Section 2, we discuss backgrounds required for the rest of this paper. In Section 3, we explain the motivation behind this work. Section 4 introduces the adaptive scheme and explains it in detail. Section 5 presents adaptive+ and discusses how the adaptive system can be enhanced using a combination of LR and decision tree. Section 6 presents experimental methodology and results. Section 7 discusses related work, and Section 8 concludes the paper.

## 2 | BACKGROUND

### 2.1 | Intel's restricted transaction memory

Intel's Haswell processor offers Transaction Synchronization Extensions (TSX), which enables programmers to write HTM programs. Haswell executes transactions on a best-effort policy, i.e., the hardware does not guarantee that a transaction will commit successfully, and thus, the programmer should provide a fall-back path for aborted transactions. TSX provides two software interfaces to execute atomic blocks: Hardware Lock Elision (HLE) and Restricted Transactional Memory (RTM). HLE is a legacy compatible programming interface for lock-based programs. RTM is a new extension in Intel's Haswell processor and enables HTM programming.

RTM offers three new instructions for HTMs: XBEGIN, XEND, and XABORT. To write an HTM program, a programmer specifies regions of a sequential program that can execute in parallel. Then, the regions are converted into transactions using the three instructions. A transaction is initiated with XBEGIN. Inside of a transaction, the underlying hardware monitors accesses to memory through the L1 data cache. The granularity in

which memory accesses are tracked is a cache line. A programmer marks the end of a transaction through the XEND instruction. This signals the hardware that any changes in the memory should be committed if there is no conflict with other transactions. To explicitly specify that a transaction should abort, XABORT is used.

In RTM, the cache coherence protocol<sup>15</sup> is in charge of conflict detection. This is necessary to avoid overhead of transactional read/write tracking in software. If two transactions access the same cache block, then cache will detect the conflict and allows only one of the two transactions to proceed; the other one should abort. RTM follows the eager policy<sup>2</sup> to detect and abort conflicting transactions. In this policy, as soon as a conflict is detected, then all conflicting transactions (except one) are aborted immediately. This is in contrast to the lazy policy<sup>2</sup> in which abort of conflicting transactions is postponed to the commit time. The eager policy improves the utilization of processor resources over the lazy policy as conflicting transactions are aborted immediately and do not occupy processor resources until commit.

There are different reasons for transactional aborts. The two most prominent causes of abort are conflicts over shared memory locations and limited capacity of the L1 cache. As mentioned earlier, the cache coherence protocol in the L1 cache is responsible for conflict detection. If the working set of a transaction exceeds the L1 cache, then it would be difficult to track transactional data. Hence, the transaction is aborted to avoid extra complexity in the cache coherence protocol. There are other sources for abort such as context switching, page fault, I/O interrupt, etc., which occur less often than conflicts and overflowed cache. The EAX register<sup>5</sup> in Haswell is a status register and shows the cause of abort. When a transaction aborts, all changes in memory are discarded and an abort code is sent to the EAX register.

## 2.2 | TinySTM

In this work, we use TinySTM<sup>3</sup> as a state-of-the-art STM. TinySTM<sup>3</sup> is chosen as it is faster than some other STM implementations such as TL2.<sup>2</sup> TinySTM is a time-based STM that efficiently constructs snapshots of memory locations accessed by a transaction. This snapshot remains consistent during the entire execution of the transaction. TinySTM offers APIs based on lock to protect shared memory locations. It also uses encounter-time locking, which is useful in the early detection of conflicts. This is in contrast to commit-time locking, which postpones lock acquisition to the commit time. Encounter-time locking improves the utilization of processor resources over commit-time locking as it avoids running a transaction that is doomed to abort.

## 2.3 | Linear regression

Linear Regression (LR)<sup>11</sup> is an approach that models the relationship between a variable and a set of input parameters using a mathematical equation. Equation 1 shows a simple model for LR:

$$y = B_0 + \sum_{k=1}^q (B_k \times x_i) + \varepsilon. \quad (1)$$

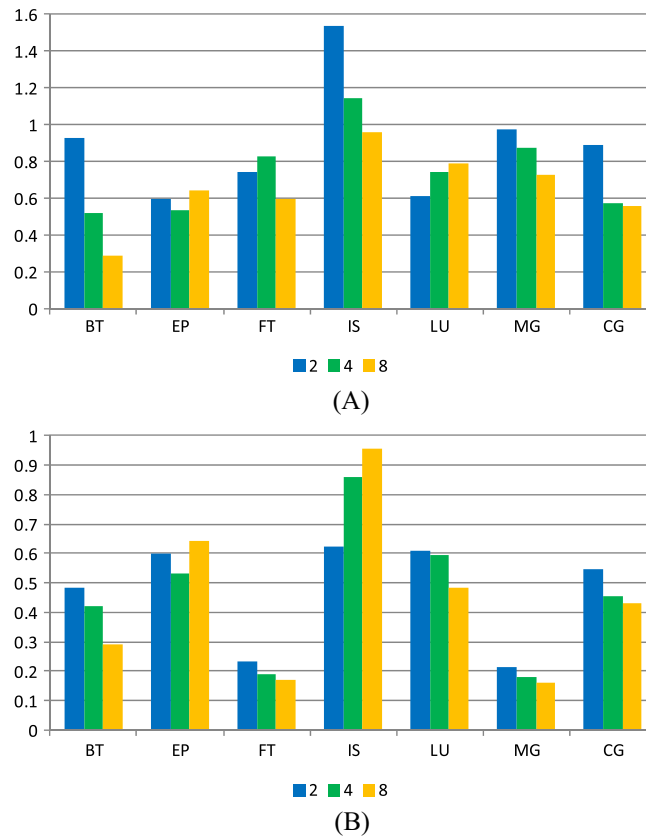
In Equation 1,  $y$  is the response variable,  $x_i$  is the input variable,  $B_0$  is the intercept of the fit with the  $y$ -axis, and  $\varepsilon$  is the error of the LR model.  $B_i$  ( $0 < i$ ) is the coefficient and shows correlation between output and input  $x_i$ . The larger the  $B_i$ , the higher the impact of  $x_i$  on the output. In LR, coefficients are found using the least squares error method. In this method, the coefficients are calculated so that the sum of the square of errors is minimized. The error of an input is defined as the distance between the predicted output corresponding to the input and the actual output. While LR uses a simple approach to relate the output variable and input parameters, it is able to predict the response variable with high accuracy. LR has been used successfully in many applications such as weather data analysis, sales of a product,<sup>11</sup> etc. Also, Google used LR to predict the revenue of a movie four weeks ahead of its release time.<sup>16</sup>

In this work, we exploit LR to predict the transaction size and reduce the execution time of transactions. To do so, we change the size of a set of transactions manually so that performance is maximized. Then, static parameters of the transactions are used as inputs in Equation 1, and the transaction size that resulted in maximum performance is used as output. Using the least squares method, coefficients in Equation 1 are calculated. Now, the equation is ready and can be used for prediction. It is important to note that while Equation 1 depends on the specific HTM and STM used in this work, our methodology is general and can be applied to other TM implementations.

## 2.4 | Decision tree

Decision tree<sup>10</sup> is a popular method for classification. In classification, a set of objects are categorized based on a set of attributes. Decision tree is used as a predictive model in many areas such as data mining, machine learning, etc. In a decision tree, there are three types of nodes: root, internal, and leaf. Classification starts from the root node and continues through the internal nodes until a leaf node is hit. The decision tree uses a test function to navigate from the root to a leaf. The input of the function is attributes of an object and the output is a binary number: 0 or 1. The test function determines whether the decision tree should navigate through the left child or the right child of a node. Each leaf node corresponds to a category.

In this work, we use decision tree to decide whether HTM or STM is a more efficient implementation for a given transaction. We also exploit decision tree to classify transactions based on the error of predicted transaction size. Objects in decision tree are transactions, and attributes of



**FIGURE 1** A, Normalized transactional execution time of RTM relative to TinySTM; B, Speedup when transaction size is optimized

the objects are transactional parameters such as read-set size, number of instructions between two consecutive transactions, etc. Sections 4 and 5 discuss details of the decision tree used in this work.

### 3 | MOTIVATION

Both RTM and TinySTM have their own advantages and restrictions, which make one of them more effective than the other for a given transaction. The main advantage of RTM over TinySTM is that its runtime overhead is less than that of TinySTM. In RTM, the cache coherence protocol is responsible for the detection and resolution of the conflicts. This reduces runtime overhead as software does not need to track transactional data. On the other side, TinySTM, similar to other STMs,<sup>2,4</sup> suffers from significant bookkeeping for data validation, conflict detection, and recovering the state of conflicted transactions. This may degrade the performance of TinySTM in some transactions. The other difference between the TMs is flexibility. RTM only relies on hardware resources for the execution of transactions. This results in complexity if hardware resources overflow due to the high volume of data (fall-back policy is needed). This may increase frequency of aborts in RTM and result in slowdown compared with TinySTM. On the other side, TinySTM does not have such resource constraints and is able to run transactions with arbitrary size. This may reduce the abort rate in large transactions and improve performance over RTM.

Figure 1A reports normalized speedup in RTM relative to STM in the NAS<sup>12</sup> benchmark suite (for details of the experimental methodology, please refer to Section 6). Bars less than 1 represent speedup under RTM. For each benchmark, the number of threads varies between two and eight. The optimum TM scheme varies across the benchmarks, primarily due to transaction characteristics such as read-set size, transaction size, etc. In small transactions, the working set of transactions fits in the data cache. As such, RTM outperforms TinySTM in these benchmarks, e.g., EP. On the other side, TinySTM is faster than RTM for large transactions, i.e., IS when the number of threads is two or four. The characteristics of transactions vary not only across applications but also within an application. We exploit this variability and propose an adaptive scheme that is able to switch between the two TM schemes within and across applications and enhance performance (Section 4).

Figure 1B shows the impact of transaction size on performance. Bars less than 1 show speedup when transaction size is optimized. We selected the fastest TM implementation for each benchmark from Figure 1A. Then, we changed the number of instructions in transactions manually and selected that which minimizes execution time. It is important to note that by changing the size of a transaction, we do not violate its atomicity. Figure 2 shows an example of a large transaction. The transaction is composed of three loops. We can decompose the outer loop into several smaller loops and assign each loop to a transaction. Similarly, we can combine smaller transactions to build a large transaction. When we change the transaction size, we take into account the atomicity of transactions to assure that we do not compromise the correctness of transactions.

```

.....
TM_BEGIN();
for(k=1;...;k++){
  for(j=1;...;j++){
    rhs_t=(double)TM_SHARED_READ_F(rhs[k][j][i][0]);
    for(i=1;i<= grid_point[0]-2;i++){
      ui=rhs_t+dx1*tx1*(up1+us1-um1);
      ....
      rhs_t=ui*tx2-u[k][j][i][1];
    }
    TM_SHARED_WRITE_F(rhs[k][j][i][0],rhs_t);
  }
}
TM_END();
.....

```

**FIGURE 2** A code snippet from BT<sup>12</sup>

Short transactions suffer from overhead of TM APIs, and so, speedup is limited in these transactions. On the other side, long transactions increase the probability of abort as the window during which transactions access shared memory locations prolongs. This increases the probability of conflicts and may hurt the performance. Hence, to boost the performance of TM programs, small transactions should be merged to reduce the overhead of APIs. On the other side, large transactions should be split to reduce the abort rate. Figure 1B shows that transaction size, indeed, has a dramatic impact on performance. In MG, performance increases up to 6.2x when we change the transaction size. On average, changing the transaction size improves performance by 53%, 54%, and 55%, when the number of threads is 2, 4, and 8, respectively.

## 4 | ADAPTIVE SYSTEM

Since neither HTM nor STM is optimum across all benchmarks, we offer an adaptive mechanism that dynamically switches between the two. We use Haswell's RTM<sup>5</sup> for HTM and a modified version of TinySTM<sup>3</sup> for STM. One factor that impacts the performance of the adaptive system is granularity of switching. One approach for switching would be using one of the two TM implementations for the entirety of an application. Such a coarse granularity approach, while simple to implement, misses many opportunities for speedup. The alternative approach would be deciding on the TM scheme per transaction. Such a fine granularity scheme exploits the characteristics of individual transactions and uses the TM scheme that results in higher speedup. In the fine granularity scheme, depending on the number of threads, a transaction might be implemented in HTM or STM. Hence, different versions of binary files should be generated. However, this extra software engineering effort is worthy as an adaptive technique is able to boost performance and reduce the energy of transactional applications (Section 6).

In the fine granularity scheme, we avoid the simultaneous execution of HTM and STM to reduce the overhead of implementation. If HTM and STM transactions execute simultaneously, then in-flight transactions should communicate to guarantee atomicity of accesses to the shared memory locations. A metadata should track accesses to transactional data, and each transaction should consult with the metadata before reading/writing transactional data. This increases the overhead of the adaptive system and restricts its scalability. In particular, an application with low contention suffers significantly from such an approach. To avoid pitfalls of simultaneous HTM-STM execution, at any moment, an application can execute either hardware or software transactions, not both.

We followed Intel's TSX manual<sup>5</sup> to implement RTM. In RTM, there is no guarantee that a transaction eventually commits in hardware. A combination of retry count and a fall-back policy determines how a transaction is treated if it fails in hardware. The retry count shows the number of times a transaction is re-executed after it aborts. There are different sources of aborts in RTM (Section 2.1). By retrying a transaction, it is possible that the source of an abort disappears (i.e., a competing transaction commits), and the transaction eventually commits. If the number of retries exceeds the retry count, then the fall-back policy is used for the execution of the transaction. We use a global lock for the implementation of the fall-back policy. The value of the retry count impacts performance. If the retry count is too high, then conflicting transactions are executed over and over. This wastes processor resources and hampers performance. On the other side, if the retry count is too low, then a transaction that could potentially commit in hardware is executed by lock. This limits transaction level parallelism and hurts performance. We evaluated different values for the retry count and found that, on average, the retry count of 4 is the best choice. The alternative would be changing the value of the retry count dynamically and based on feedback received from executing benchmarks. However, this method is beyond the scope of this paper.

### 4.1 | Synchronization of RTM and STM

Our adaptive system executes a subset of transactions in RTM mode and the rest in STM mode. As explained in the previous section, to avoid complexity, at any moment, transactions execute either in RTM mode or in STM mode but not in both. To support the mutual exclusion of RTM and STM,

```

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

```

**FIGURE 3** Pseudo code for synchronization of HTM and STM in tx\_start()

we use conditional variables. Figures 3 and 4 show the pseudo code for synchronization. Conditional variables are bolded. Mutual exclusion is very crucial since if there is any issue with its implementation, then some synchronization bugs such as deadlock or livelock may happen. Also, transactions may update shared memory locations simultaneously, which may result in incorrect execution of applications.

The synchronization is implemented inside tx\_start() and tx\_commit() where a transaction starts and ends, respectively. Each of the two functions receives an input argument: htm\_n\_stm. This variable shows whether the corresponding transaction should execute in HTM or STM. A value of 1 represents HTM; otherwise, STM. The synchronization in Figure 3 has two similar parts: lines 4-12 and 14-22. The first part corresponds to HTM mode, and the second part corresponds to STM mode. Since the two parts are similar, we only explain the first part.

When a transaction starts in HTM mode, it checks the number of in-flight STM transactions (line 7). If there are some STM transactions executing simultaneously, then the transaction waits (line 8). This is necessary to guarantee mutual exclusion. Once, there is no STM transaction, the transaction proceeds and increments a num\_in\_flight\_htm variable. This variable, as the name shows, represents the number of in-flight transactions in HTM mode. A global lock (htm\_stm\_sync\_mutex) is used to guarantee the atomicity of global variables used during synchronization. It is important to note that the amount of time spent in the critical section depends on the sequence of executing transactions. If a software transaction is followed by a hardware transaction and before all instances of the software transaction commit, a hardware transaction starts, then the hardware transaction should wait. On the other side, if the hardware transaction starts after all instances of the software transaction commit, then the hardware transaction does not wait in the critical section.

The synchronization in tx\_commit() has two similar parts: one for HTM and the other for STM. We only explain the first part due to the similarity of the two sections. When a transaction reaches the commit stage, it decrements num\_in\_flight\_htm (line 31). Access to the variable is protected by the global lock. If this variable is zero, then a signal is broadcasted to all waiting transactions (line 33). This is necessary to wake up all those transactions that are waiting for in-flight hardware transactions to finish (line 18). Failure to do so results in deadlock.

In Figures 3 and 4, the lock (htm\_stm\_sync\_mutex) is accessed in two functions: tx\_start() and tx\_commit(). In each of these two functions, the lock is acquired and then released. So, the lock never causes deadlock as no thread can hold the lock indefinitely. In tx\_commit(), a thread decrements the counter for the number of in-flight HTM/STM transactions, wakes up transactions that are waiting for the conditional variables, and then exits the critical section. So, tx\_commit() does not cause deadlock. To prove that the adaptive technique is deadlock free, we only need to focus on tx\_start().

Since the way HTMs and STMs are treated in tx\_start() is similar, we only prove that tx\_start() is deadlock free for HTMs. When an HTM starts, if there is no in-flight STM in the system, then the HTM exits the critical section. However, if there is at least one in-flight STM, then the HTM waits

```

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

```

**FIGURE 4** Pseudo code for synchronization of RTM and STM in tx\_commit()

over the conditional variable (line 8). When the last STM in the system reaches commit, it sets the number of in-flight STMs to zero (line 40) and wakes up the HTM (line 42). So, the HTM exits the critical section in tx\_start().

## 4.2 | Implementation of decision tree prediction module

In this work, we exploit decision tree to predict whether HTM or STM is a better choice for a given transaction. To generate the decision tree from a training dataset, we use the C4.5 algorithm.<sup>17</sup> This algorithm improves the accuracy of classification over the previous ID3 algorithm<sup>17</sup> by dealing with both continuous and discrete variables and pruning the decision tree after construction.

For training of the decision tree, we use four transactional parameters: transaction size, read-set size, write-set size, and number of threads. These parameters are important as they characterize a transaction. The output of the decision tree is a binary number and shows whether the transaction should be executed in HTM or STM mode. These parameters are generated through profiling. We changed TinySTM so that it automatically generates these parameters when a benchmark executes.

To train the decision tree, we converted the NAS benchmark suite<sup>12</sup> into transactional applications. The set of transactions in the NAS are composed of small, medium, and large size transactions. Each benchmark is executed twice: once in RTM mode and once in STM mode. Then, the decision tree is trained based on statistics generated through the two rounds of execution. This procedure was repeated for 2, 4, and 8 number of threads as the characteristics of transactions may vary with the number of threads. Appendix B shows the decision tree used in the adaptive scheme.

To provide a better insight into the training of the decision tree, we provide parameters associated with transactions in the IS benchmark<sup>12</sup> (Table 1). This benchmark has seven transactions, and each transaction has its own characteristics. One of the inputs to the decision tree is transaction size. One way to measure the size of a transaction (the sixth column in Table 1) is counting the number of C code lines in a transaction. However, this method may not truly distinguish transactions with regard to their execution times as execution time varies across lines of a C program. A more effective way is counting the number of assembly instructions corresponding to a transaction. We used this method for training.

As explained earlier, RTM is faster than STM for small transactions. Benchmark IS has four transactions with sizes ranging 1141-10752. For these transactions, RTM executes faster than STM. The size of other transactions is equal to or more than 92224 lines. For these transactions, STM outperforms RTM by a large margin.

**TABLE 1** Characteristics of benchmark IS<sup>12</sup> consisting of seven transactions

TX #	STM Time, ms	RTM Time, ms	Read-Set Size	Write-Set Size	TX Size	Write Ratio
TX1	0.774	0.629	256	256	10752	0.5
TX2	16.21	24.345	16384	16384	688128	0.5
TX3	0.066	0.052	7	7	1141	0.5
TX4	1.373	1.452	128	128	92224	0.5
TX5	0.197	0.134	12	12	2748	0.5
TX6	17.721	33.364	18225	18630	264951	0.5
TX7	0.595	0.541	81	81	8262	0.5

## 5 | ADAPTIVE+ SYSTEM

In this section, we propose adaptive+, which enhances the performance of the adaptive scheme by optimizing hardware and software transactions. In particular, we focus on the transaction size, which has a significant impact on performance. While small transactions suffer from overhead of the TM system, large transactions may face slowdown due to cost of aborts. One way to optimize the transaction size is using a try-and-error approach. However, this method is time consuming and is not feasible for real-world applications with a large number of transactions and different transaction sizes. To automate this process, we build a linear regression model that predicts transaction size. The main reason that we selected transaction size for the optimization of the adaptive system is that it requires moderate software engineering effort. Quite often, it does not require changing the data structure of a program (i.e., the code snippet in Figure 2). It is important to note that not all transactions can benefit from this technique. For example, if a transaction is composed of a loop and loop iterations are dependent, then reducing the transaction size by breaking down the original loop into smaller loops is not feasible.

Since the characteristics of RTM and STM are different, we build LR models for RTM and STM, separately. We use SPSS<sup>18</sup> to find coefficients in our LR models. We investigated different combinations of inputs for LR and concluded that we need five extra inputs in addition to size of transaction (ST), size of write-set (WS), and size of read-set (RS), which were used as inputs for the decision tree. The additional parameters are size of next transaction (SNT), number of assembly instructions between two consecutive transactions (NCT), write-set of the next transaction (WN), read-set of the next transaction (RN), and number of assembly instructions in a loop (TL). We use profiling to generate these parameters. Next, we will explain the intuition behind choosing the additional five inputs.

- a) SNT: If the size of a transaction is less than the size predicted by LR, then we may need to merge the transaction with its successive transaction and build a large one. For example, assume that transaction A with size 3000 is followed by transaction B with size 5000. Assuming that the predicted size is 8000, it is possible to merge the two transactions. However, if the sizes of transactions A and B are 4000 and 6000, respectively, then the two transactions cannot be merged as the combined transaction has much more than 8000 instructions.
- b) NCT: NCT impacts the merging of transactions when two successive transactions should be merged and there are non-transactional instructions between the two. For example, assume that there are two successive transactions, each with 3000 instructions, and there are 2000 non-transactional instructions between the two. Similar to the previous example, assume that the predicted transaction size is 8000. It is possible to merge the two transactions and also the non-transactional instructions between the two and build a transaction with 8000 instructions. However, if NCT is 5000, then it is not possible to merge the two transactions as the combined transaction is too large and hurts performance.
- c) WN and RN: Similar to SNT, write-set and read-set of the next transaction may determine the feasibility of merging two successive transactions. So, to optimize transaction size, we need to consider WN and RN as well.
- d) TL: This parameter is important only in those transactions that are inside the body of a loop. If the size of a loop iteration is less than the predicted transaction size, then the entire loop can be moved into a transaction. For those transactions that do not have any loop, we set this parameter to zero.

Further investigation of the LR model reveals that the error rate of LR predictions varies. In some transactions, the error rate is a large positive number. On the other side, in some others, the error is a large negative number. This motivates us to categorize transactions into two groups: transactions with large negative error (group1) and transactions with large positive error (group2). To increase the accuracy of predictions, we use separate LR models for each group. This improves the accuracy of LR predictions as within each group, the set of points are well organized and it is possible to fit a line to the set of points with less residual error. As mentioned earlier, we use separate LR models for RTM and STM since the two TM schemes have different characterizations. Also, we customize the LR model for different numbers of threads. Appendix A shows the details of LR models used in this work.

To be able to use LR models, we need a mechanism to specify the group that a transaction belongs to. This is a classification problem, and we use the decision tree for this purpose (Appendix B). We use transactions from NAS for the training of the decision tree. Each training sample has an 8-dimensional input vector (SNT, NCT, WN, RN, TL, ST, WS, and RS) and an output that indicates the group the transaction belongs to. Once the decision tree is trained, we use it for prediction. For a given transaction, we feed its 8 transactional parameters to the decision tree and the tree predicts its corresponding group. Then, the LR model of the group (Appendix A) is selected to predict the size of the transaction.

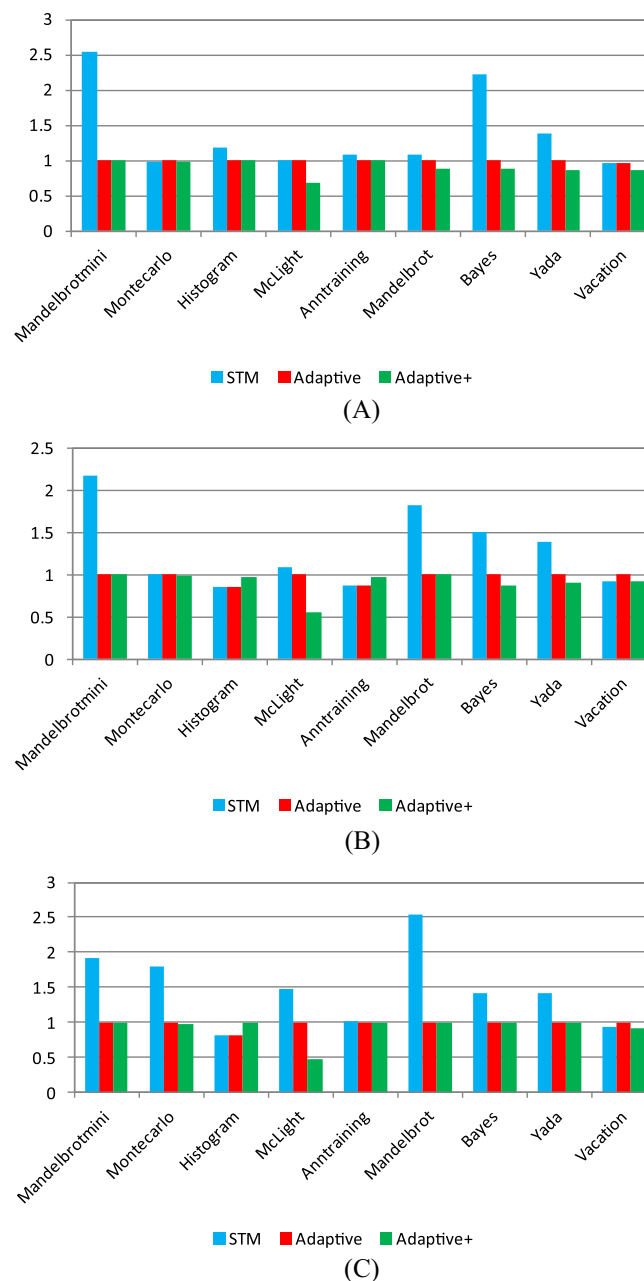


## 6 | EXPERIMENTAL RESULTS

For consistency, we use the same hardware platform to evaluate both RTM and STM. The experimental setup includes an Intel 4th Generation Core™. The processor consists of four cores with two hyper threads per core, for a total of eight threads. Each core has a 32-KB L1 instruction cache, a 32-KB L1 data cache, and a 256-KB L2 cache. The four cores share an 8-MB L3 cache. We use GCC v4.8.1 to compile benchmarks and calculated average of statistics over 10-run. To enable Intel's TSX intrinsic, we compile benchmarks with `-mrtm` flag.

To evaluate an STM system, researchers rely on a set of benchmarks. If the set of benchmarks are selected from a specific field, then the outcome of the research is not reliable. To be able to extend the outcome of a research project to real-world applications, we need a set of comprehensive benchmarks that truly represent real-world applications. Asanovic et al.<sup>19</sup> proposed 13 Dwarfs as a guideline to develop benchmark suites for parallel applications. A dwarf is a high-level abstraction that categorizes applications based on patterns of computation and communication. Asanovic et al.<sup>19</sup> showed that the NAS benchmark suite<sup>12</sup> includes all those dwarfs, and so, in this work, we use the NAS benchmark suite to train LR and decision tree.

We use the DiscoPop<sup>13</sup> and STAMP<sup>14</sup> benchmark suites to measure performance and energy of adaptive+. Since we are using different sets of benchmarks for training and testing, we avoid overfitting in our prediction models. Figure 5 presents the performance of STM, adaptive, and adaptive+ normalized to the performance of RTM. A benchmark that consists of a value more than 1 shows speedup under RTM. On average, adaptive and



**FIGURE 5** Normalized speedup. Bars more than 1 represent speedup under RTM. A, 2-thread; B, 4-thread; C, 8-thread

**TABLE 2** Predicted and optimum transaction size

Name	Original TX Size	Applied TX Size Based on Prediction	Optimum TX Size
Mandelbrot_mini	51300	12312	10260
Montecarlo	85261	8525	8525
Histogram	320625	9672	9672
McLight	116250	10130	10130
Anntraining	28800	11772	11772
Mandelbort	78208	18096	16334
Bayes	1261	2687	2687
Yada	573	1859	1859
Vacation	672	1429	1429

adaptive+ improve performance over RTM by 3% and 9%, respectively. In some benchmarks, i.e., Bayes, RTM is significantly faster than STM. Using decision tree, the adaptive scheme is able to select the better of the two and enhance performance over STM. In some benchmarks, i.e., McLight, adaptive+ has a significant speedup over the adaptive system. In these benchmarks, adaptive+ exploits LR to optimize transaction size. This reduces the overhead of TM and enhances the performance of TM over the adaptive system.

Table 2 shows the transaction size predicted by decision tree and LR as well as the optimum transaction size. The optimum transaction size is calculated by manually changing transactions and evaluating the impact of each transaction on performance. It is important to note that in some benchmarks, we had to change the predicted transaction size based on benchmarks' constraints. For example, if a transaction is within a loop and instructions in the loop cannot be changed due to dependency, then transaction size should be multiple of loop size. In the majority of the benchmarks, predicted and optimum transaction sizes are the same. On average, the accuracy of predictions is 97%.

## 6.1 | Energy expenditure analysis

While most of research articles in TM focus on performance, energy of TM applications is an important factor that should be optimized. In particular, many computing devices such as laptop, tablets, etc., rely on battery power. As such, low-power processors are essential to increase battery lifetime. We use Intel's Runtime Average Power Limit (RAPL)<sup>20</sup> to read energy of processor package. RAPL offers a set of APIs to access hardware counters and read energy and power consumption of processors.

Figure 6 presents energy of the test benchmarks. We report energy of the entire applications. Bars in Figure 6 are normalized to the energy of RTM (readings of more than 1 show energy reduction under RTM). STM consumes more energy than RTM as it relies on software for the execution of transactions. On the other side, RTM exploits hardware resources and, so, is more energy efficient than STM. On average, STM increases energy of applications by 33%.

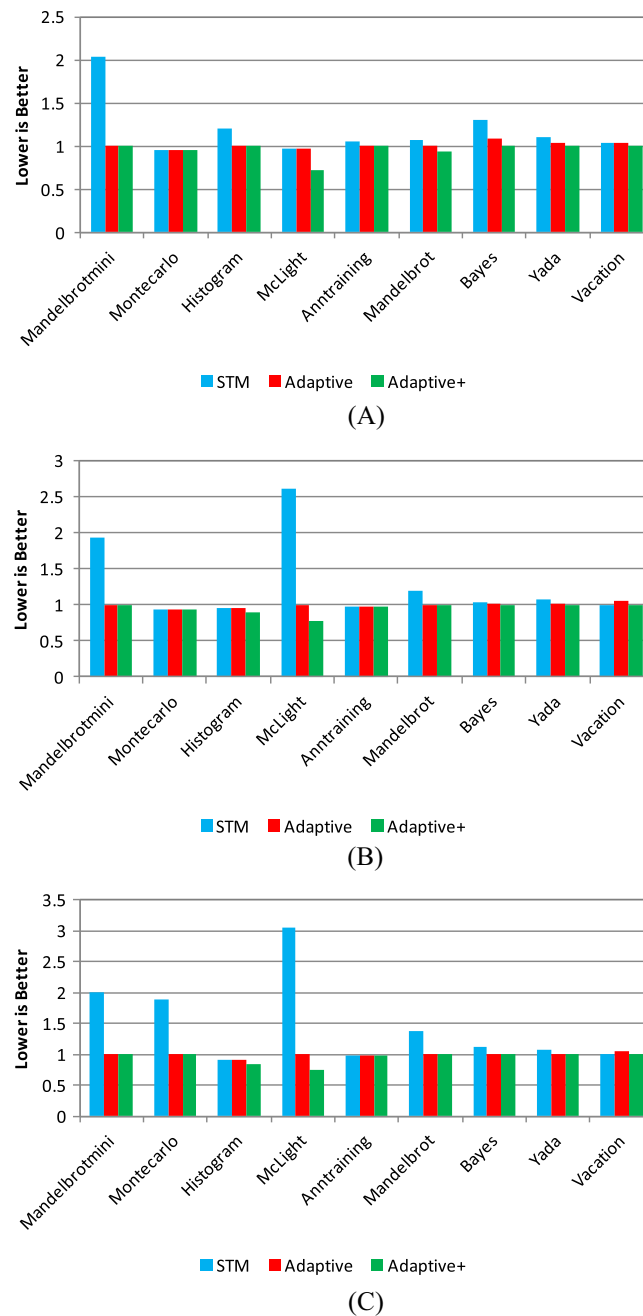
While in some benchmarks, i.e., Bayes, the adaptive scheme falls behind RTM, in some others, i.e., Montecarlo, the adaptive scheme improves energy efficiency over RTM. On average, energy of benchmarks is similar under adaptive and RTM schemes. Adaptive+ improves energy over adaptive as adaptive+ optimizes transaction size and reduces overhead of transactions. Across all benchmarks, energy of adaptive+ is less than or similar to the energy of RTM. On average, adaptive+ reduces energy by 7% over RTM. This reduces the energy-delay of adaptive+ by 15% over RTM.

## 7 | RELATED WORK

Transactional Memory was initially proposed as an extension to the cache coherence protocol in shared memory multiprocessors.<sup>21</sup> However, due to the unavailability of a hardware platform to support such a TM, researchers switched to STM to advance the state-of-the-art TM systems.<sup>2</sup> In STMs, a programmer or a compiler instruments the code to invoke STM APIs when a shared memory location is accessed. This may result in extra runtime overhead. To cope with the overhead of STM, chip vendors such as Intel<sup>5</sup> and IBM<sup>6,7</sup> delivered processors that provide hardware support for TM. We evaluated adaptive+ using only an Intel processor as we did not have access to other processors. Nevertheless, we believe that adaptive+ is independent of processor vendors as processors with HTM support have a similar nature.

Yoo et al.<sup>22</sup> evaluated Intel's TSX using a set of high-performance computing workloads. RTM is able to avoid unnecessary serialization, which increases concurrency and improves scalability. Furthermore, RTM reduces the cost of synchronization when there is no contention. This improves performance. Wang et al.<sup>23</sup> extended the aforementioned work<sup>22</sup> by evaluating the impact of some transactional parameters such as transaction size, write-ratio, etc., on performance. They used microbenchmarks for evaluation and compared RTM with locks. Our work is different as we exploit both RTM and STM to improve the performance of transactional applications.

Pereira et al.<sup>24</sup> evaluated the performance of RTM extensively. In particular, they focused on RTM's forward-progress policies as in RTM, it is not guaranteed that a transaction will eventually commit. This work proposes retrying a transaction certain number of times in an attempt to commit the transaction. In particular, they propose SerControl, which decides on retrying a transaction based on the type of transactional abort. SerControl



**FIGURE 6** Normalized energy. Bars more than 1 represent energy saving under RTM. A, 2-thread; B, 4-thread; C, 8-thread

uses status bits in EAX register to find the cause of abort. If a transaction aborts due to conflict or capacity, then SerControl avoids running the transaction concurrently by serializing the execution of the transaction through a lock; otherwise, the transaction is serialized after retrying the transaction for certain number of times. Pereira et al.<sup>24</sup> do not compare their proposed forward-progress scheme with another STM. We propose adaptive+, which dynamically decides whether an STM or RTM implementation should be used for a given transaction. If a transaction is long or it has a large working set, no matter which policy is used for forward progress, RTM falls behind STM.

DiscoPoP<sup>13</sup> is a tool that receives a sequential program as input and generates a code that shows parallelizable sections of the input. DiscoPoP analyzes the input code and identifies code regions that do not have any true data dependency and calls it Compute Units (CUs). Then, it creates a graph where its nodes are CUs and edges represent dependency between CUs. From this graph, it is possible to derive potential parallelism available in varying levels of the code including nested regions. To be precise, DiscoPoP generates a code that shows lines of a sequential program that can be grouped and run in parallel with other groups. CUs never cross the boundary of control instructions. As such, DiscoPoP is able to detect parallelism among higher-level constructs such as loops or functions. We used microbenchmarks introduced in the DiscoPoP paper to evaluate adaptive+. DiscoPoP can be combined with our proposed scheme to automatically convert sequential programs into optimized transactional programs.

Didona et al.<sup>25</sup> investigated the concurrency level in STMs. The performance of parallel applications depends on the degree of parallelism among executing tasks. If too many tasks execute in parallel, there will be significant contention over shared memory locations. This increases the abort rate

and reduces performance. On the other side, a few parallel tasks reduce the utilization of processor resources and hurt performance. The right level of parallelism depends on many parameters in both hardware and software, including but not limited to size of memory hierarchy, OS scheduler, etc. Didona et al.<sup>25</sup> used a hill-climbing algorithm to identify the appropriate level of parallelism in STMs. This approach can be combined with adaptive+ to adjust transaction level parallelism dynamically and improve performance further.

Xiao et al.<sup>26,27</sup> focused on STMs and showed that the performance of STMs is sensitive to transaction size. Jeyakumaran et al.<sup>28,29</sup> proposed an adaptive technique that dynamically switches between RTM and STM. This work is an extension of our previous works on TMs.<sup>26-29</sup> We enhance the performance of TM systems by introducing adaptive+. Adaptive+ goes through two phases to reduce the overhead of TM systems. In the first phase, it decides whether HTM or STM should be used for a given transaction. In the second phase, it adjusts the transaction size based on some transactional parameters such as read-set size, write-set size, etc. Most of the research articles on TMs focus only on performance. In this work, in addition to performance, we evaluated TM from the energy point of view and demonstrated that adaptive+ indeed reduces energy of TM applications.

Castro et al.<sup>30,31</sup> exploited machine learning algorithms to map threads to processing cores dynamically. In particular, thread mapping targets memory hierarchy and tries to reduce latency of memory accesses. This is accomplished through training of decision tree at specific intervals. During each interval, the decision tree is trained, and then, in the following interval, it predicts thread mapping using the ID3 algorithm.<sup>17</sup> We exploited the decision tree to decide whether HTM or STM is a better choice for a given transaction. We used the C4.5 algorithm for the decision tree instead of ID3 as it is able to support continuous parameters, which results in more accurate predictions.

Wang et al.<sup>32</sup> investigated the impact of workload characteristics on STMs and proposed an adaptive mechanism to adjust transactional parameters. There are diverse algorithms for the implementation of STMs. Each of these algorithms works well under certain runtime systems and for certain workloads. Hence, there is no single STM algorithm that works well across all applications. Depending on the characteristics of executing transactions such as ratio of read-only transactions, frequency of irrevocable transactions, etc., an STM may work better than the other. Wang et al.<sup>32</sup> exploited profiling to measure the behavior of a running program and change the STM algorithm to improve performance. We used TinySTM for the implementation of STM in adaptive+. This work can be used to change the configuration of TinySTM dynamically and enhance the performance of adaptive+ further.

## 8 | CONCLUSION

In this paper, we show that depending on transaction characteristics, HTM or STM is a better choice for execution of a given transaction. We exploit this property and propose an adaptive system that seamlessly switches between RTM and TinySTM. We use decision tree to classify transactions and run transactions in either HTM or STM mode. We further improve the performance of the adaptive scheme by optimizing individual transactions. In particular, we focus on transaction size and use an LR model to predict the optimum transaction size based on eight transactional parameters. To improve the accuracy of predictions, we use different LR models based on the error of predictions as well as the number of threads. To select an appropriate LR model, we use decision tree. Our proposed adaptive+ scheme is able to improve both performance and energy significantly.

### ORCID

Ehsan Atoofian  <http://orcid.org/0000-0002-1662-5334>

### REFERENCES

1. Sutter H, Larus JR. Software and the concurrency revolution. *Queue*. 2005;3(7):54-62.
2. Dice D, Shalev O, Shavit N. Transactional locking II. Paper presented at: 20th International Symposium on Distributed Computing; 2006; Stockholm, Sweden.
3. Felber P, Fetzer C, Riegel T. Dynamic performance tuning of word-based software transactional memory. Paper presented at: 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming; 2008; Salt Lake City, UT.
4. Saha B, Adl-Tabatabai A-R, Hudson RL, Chi Cao M, Hertzberg B. McRT-STM: a high performance software transactional memory system for a multi-core runtime. Paper presented at: 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP); 2006; New York, NY, USA.
5. Intel Corporation Chapter 12. Intel's Transactional Synchronization Extensions (TSX). 2013. <http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-optimization-manual.html>
6. Wang A, Gaudet M, Wu P, et al. Evaluation of Blue Gene/Q hardware support for transactional memories. Paper presented at: 21st International Conference on Parallel Architectures and Compilation Techniques, PACT'12; 2012; New York, NY.
7. Jacobi C, Slegel T, Greiner D. Transactional memory architecture and implementation for IBM System z. Paper presented at: International Symposium on Microarchitecture, MICRO'12. IEEE Computer Society; 2012; Vancouver, BC, Canada.
8. Hammond L, Wong V, Chen M, et al. Transactional memory coherence and consistency. Paper presented at: Proceedings of the 31st International Symposium on Computer Architecture (ISCA); 2004; Munich, Germany.
9. Moore KE, Bobba J, Moravan MJ, Hill MD, Wood DA. LogTM: log-based transactional memory. Paper presented at: 12th International Conference on High-Performance Computer Architecture (HPCA); 2006; Austin, TX.
10. Quinlan JR. Induction of decision tree. *Mach Learn*. 1986;1(1):81-106.
11. Goldberger S. Best linear unbiased prediction in the generalized linear regression model. *J Am Stat Assoc*. 1962;57(298):369-375.

12. Bailey D, Barszcz E, Barton J, et al. The NAS parallel benchmarks. RNR Technical Report RNR-94-007; 1994.
13. Zhen L, Jannesari A, Wolf F. Discovery of potential parallelism in sequential programs. Paper presented at: 42nd International Conference on Parallel Processing Workshops (ICPPW), Workshop on Parallel Software Tools and Tool Infrastructures (PSTI); 2013; Lyon, France.
14. Minh CC, Chung J, Kozyrakis C, Olukotun K. STAMP: stanford transactional applications for multi-processing. Paper presented at: IEEE International Symposium on Workload Characterization (IISWC); 2008; Seattle, WA, USA.
15. Culler DE, Singh J, Gupta A. *Parallel Computer Architecture: A Hardware/Software Approach*. San Francisco, CA: Morgan Kaufmann Publishers; 1999.
16. Google Inc. Quantifying Movie Magic with Google Search. 2013
17. Quinlan JR. *C4.5: Programs for Machine Learning*. San Francisco, CA: Morgan Kaufmann Publishers; 1993.
18. SPSS Inc. *SPSS 16.0 Command Syntax Reference*. Chicago, IL: SPSS Inc.; 2007.
19. Asanovic K, Bodik R, Catanzaro BC, et al. The Landscape of Parallel Computing Research: A View From Berkeley. Technical Report UCB/EECS-2006-183. Berkeley, CA: EECS Department, University of California; 2006.
20. *Intel Architecture Software Developer's Manual: System Programming Guide*; June, 2013.
21. Herlihy M, Moss JEB. Transactional memory: architectural support for lock-free data structures. Paper presented at: 20th Annual International Symposium on Computer Architecture (ISCA); 1993; San Diego, CA, USA.
22. Yoo RM, Hughes CJ, Lai K, Rajwar R. Performance evaluation of Intel transactional synchronization extensions for high-performance computing. Paper presented at: International Conference on High Performance Computing, Networking, Storage and Analysis; 2013; New York, NY.
23. Wang MD, Burcea M, Li L, Sharifymoghaddam S, Steffan G, Amza C. Exploring the performance and programmability design space of hardware transactional memory. Paper presented at: The ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT); March, 2014; Raleigh, NC, USA.
24. Pereira MM, Gaudet M, Amaral JN, Araújo G. Multi-dimensional evaluation of Haswell's transactional memory performance. Paper presented at: 2014 IEEE 26th International Symposium on Computer Architecture and High Performance Computing; 2014; France.
25. Didona D, Felber P, Harmanci D, Romano P, Schenker J. Identifying the optimal level of parallelism in transactional memory applications. In: Gramoli V, Guerraoui R, eds. *NETYS 2013. LNCS*. Vol. 7853. Heidelberg: Springer; 2013:233-247.
26. Xiao Y, Li Z, Atoofian E, Jannesari A. Automatic optimization of software transactional memory through linear regression and decision tree. Paper presented at: 15th International Conference on Algorithms and Architectures for Parallel Processing; 2015; China.
27. Yang X. Optimization of Software Transactional Memory Through Linear Regression and Decision Tree [Master's Thesis]. Thunder Bay, ON, Canada: Lakehead University; 2016.
28. Jeyakumar T, Atoofian E, Xiao Y, Li Z, Jannesari A. Improving performance of transactional applications through adaptive transactional memory. Paper presented at: 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing; 2016; Greece.
29. Jeyakumar T. Improving Performance of Transactional Applications Through Adaptive Transactional Memory [Master's Thesis]. Thunder Bay, ON, Canada: Lakehead University; 2016.
30. Castro M, Góes LFW, Méhaut J-F. Dynamic thread mapping based on machine learning for transactional memory applications. Paper presented at: Euro-Par; 2012; Rhodes Island, Greece.
31. Castro M, Góes LFW, Méhaut J-F. Adaptive thread mapping strategies for transactional memory applications. *Int J Parallel Distrib Comput (JPDC)*. 2014;74:2845-2859.
32. Wang Q, Kulkarni S, Cavazos J, Spear M. Towards applying machine learning to adaptive transactional memory. *TRANSACT*. 2011.

**How to cite this article:** Xiao Y, Jeyakumar T, Atoofian E, Jannesari A. Improving performance of transactional memory through machine learning. *Concurrency Computat: Pract Exper*. 2017;e4397. <https://doi.org/10.1002/cpe.4397>

## APPENDIX A

This section presents the LR model used for HTM and STM. To increase the accuracy of predictions, we use separate models for different numbers of threads.

### 2 Threads – HTM:

$$TS_1 = 2960.557 + 0.225 \times ST - 3.17 \times WS + 0.004 \times TL - 0.073 \times NCT - 0.018 \times SNT - 5.806 \times RN$$

$$TS_2 = 15592.312 - 0.403 \times ST - 1.924 \times WS + 0.182 \times TL - 1.924 \times NCT + 9.79 \times SNT - 0.18WR$$

### 4 Threads – HTM:

$$TS_1 = 3249.904 + 0.209 \times ST - 2.953 \times WS + 0.006 \times TL - 0.069 \times NCT - 0.046 \times SNT - 3.914 \times RN$$

$$TS_2 = 27.869 + 0.131 \times ST + 420.849RS - 420.784 \times WS + 0.869 \times TL + 867.372 \times NCT - 1.719 \times SNT - 626.514 \times WN$$

**8 Threads - HTM:**

$$TS_1 = 2183.844 + 0.078 \times ST - 1.043 \times WS + 0.005 \times TL - 0.044 \times NCT + 0.008 \times SNT - 2.541 \times RN$$

$$TS_2 = 15902.1 - 1.469 \times ST + 217.5RS - 2522.793 \times RN + 1.254WN$$

**2 Threads - STM:**

$$TS_1 = 5023.711 + 0.173 \times ST - 2.454 \times WS + 0.081 \times TL + 0.867 \times NCT - 0.058 \times SNT - 2.379 \times WN$$

$$TS_2 = 1757.887 + 0.824 \times ST - 53.746 \times WS - 1.55 \times TL - 2.287 \times SNT + 554.253 \times RN$$

**4 Threads - STM:**

$$TS_1 = 3482.73 + 0.177 \times ST - 2.328 \times WS + 0.011 \times TL - 0.078 \times NCT - 0.045 \times SNT - 3.225 \times WN$$

$$TS_2 = -107.488 + 1.254 \times ST - 11.499 \times WS - 0.356 \times TL - 998.429 \times NCT - 4.012 \times SNT + 854.345 \times RN$$

**8 Threads - STM:**

$$TS_1 = 3788.654 + 0.2 \times ST - 2.503 \times WS + 0.014 \times TL - 0.09 \times NCT - 0.057 \times SNT - 3.602 \times RN$$

$$TS_2 = 1037.249 + 0.992 \times ST - 434.69 \times WS + 0.1 \times TL - 16.695 \times NCT - 0.009 \times SNT + 437.505 \times RN$$

**APPENDIX B**

This section shows the decision tree that is used in the adaptive scheme. The decision tree determines whether HTM or STM should be used for the implementation of a transaction.

ReadSetSize  $\leq$  27 :

| transactionSize  $\leq$  2771 : rtm.

| transactionSize  $>$  2771 :

| | numberOfthreads = 2 : rtm.

| | numberOfthreads = 8 : rtm.

| | numberOfthreads = 4 :

| | | WriteSetSize  $\leq$  12 : stm.

| | | WriteSetSize  $>$  12 : rtm.

ReadSetSize  $>$  27 :

| transactionSize  $>$  3870 : stm.

| transactionSize  $\leq$  3870 :

| | lengthToNext  $>$  679 : stm.

| | lengthToNext  $\leq$  679 :

| | | nextRd  $>$  128 : stm (3.0).

| | | nextRd  $\leq$  128 :

| | | | numberOfthreads = 2 : rtm.

| | | | numberOfthreads = 4 : stm.

| | | | numberOfthreads = 8 :

| | | | | lengthOfNext  $\leq$  3271 : stm.

| | | | | lengthOfNext  $>$  3271 : rtm

**APPENDIX C**

This section shows the decision trees used to classify transactions based on the error of LR. HTM and STM use different decision trees. Also, depending on the number of threads, we use a different decision tree.

**2 Threads - HTM:**

transactionSize  $\leq$  8262 : LR1.

transactionSize  $>$  8262 :

| transactionSize  $\leq$  55674 : LR2.

| transactionSize >55674 : LR1.

#### **2 Threads - STM :**

WriteSetSize >120 : LR1.

WriteSetSize <= 120 :

| totalLoop <= 0 : LR1.

| totalLoop >0 : LR2.

#### **4 Threads - HTM :**

transactionSize <= 5376 : LR1.

transactionSize >5376 :

| transactionSize >84630 : LR1.

| transactionSize <= 84630 :

| | nextRd <= 96 : LR2.

| | nextRd >96 : LR1.

#### **4 Threads - STM :**

totalLoop <= 0 : LR1.

totalLoop >0 :

| transactionSize >23296 : LR1.

| transactionSize <= 23296 :

| | totalLoop <= 92224 : LR2.

| | totalLoop >92224 : LR1.

#### **8 Threads - HTM**

transactionSize <= 8262 : LR1.

transactionSize >8262 :

| lengthOfNext <= 918 : LR2.

| lengthOfNext >918 : LR1.

#### **8 Threads - STM :**

totalLoop <= 0 : LR1.

totalLoop >0 :

| ReadSetSize <= 32 : LR2.

| ReadSetSize >32 :

| | lengthOfNext <= 8262 : LR1.

| | lengthOfNext >8262 :

| | | totalLoop <= 264951 : LR2.

| | | totalLoop >264951 : LR1.