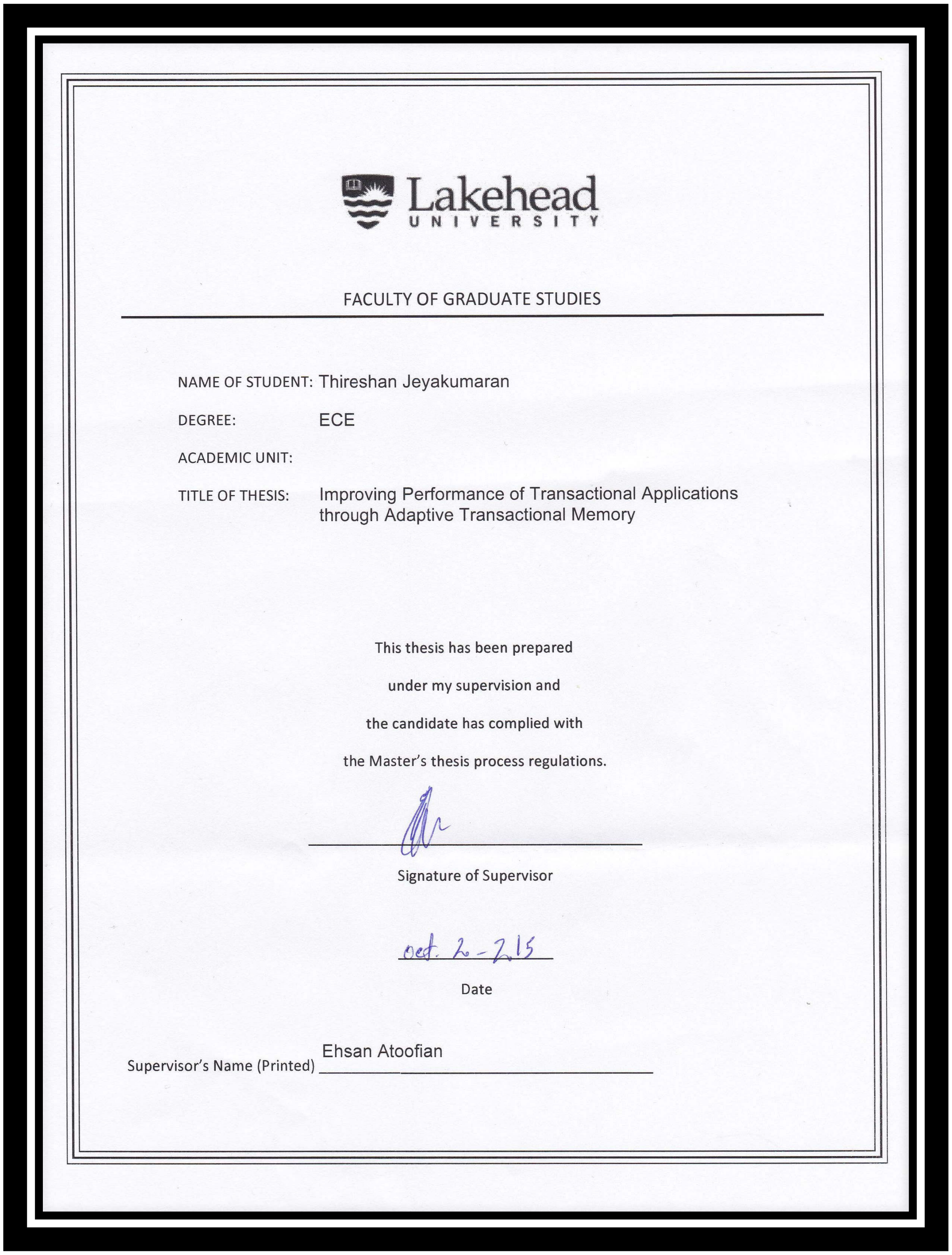
Improving Performance of Transactional Applications through Adaptive Transactional Memory

Thireshan Jeyakumaran

Supervisor: Dr. Ehsan Atoofian

Department of Electrical and Computer Engineering
Lakehead University

Thunder Bay, Ontario, Canada
Sept, 2015



Lakehead UNIVERSITY

FACULTY OF GRADUATE STUDIES

NAME OF STUDENT:	Thireshan Jeyakumaran
DEGREE:	ECE
ACADEMIC UNIT:	
TITLE OF THESIS:	Improving Performance of Transactional Applications through Adaptive Transactional Memory

This thesis has been prepared

under my supervision and

the candidate has complied with

the Master's thesis process regulations.

[illegible]

Signature of Supervisor

oct. 2-2015

Date

Supervisor's Name (Printed) Ehsan Atoofian

Abstract

With the rise of chip multiprocessors (CMPs), it is necessary to use parallel programming to exploit computational power of CMPs. Traditionally, lock-based mechanisms have been used to synchronize shared variables in parallel programs. However, with the complexity associated with locks, writing a correct parallel program is a huge burden for programmers. As an alternative, Transactional Memory (TM) is gaining momentum as a parallel programming model for multi-core processors. TM provides programmers with an atomic construct (transaction), which can be used to guarantee atomicity of accesses to shared variables, as the synchronization is handled through the underlying system. Transactional memory comes in two variants: Software transaction memory (STM) and Hardware transaction memory (HTM). Both STM and HTM systems have advantages and disadvantages that either enhance or penalize performance in transactional applications.

In this thesis, the focus is on implementing an adaptive system that exploits both STM and HTM at transaction granularity. The goal is to achieve performance gain by incorporating the benefits of both TM systems. A synchronization technique is developed to seamlessly switch between HTM and STM based on the characteristics of a transaction. We exploit 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 transactions based on parameters such as transaction size, transaction write ratio, etc. From the evaluations using STAMP, NAS, and DiscoPoP benchmark suites, the proposed adaptive system is able to improve speed of transactional applications by 20.82% on average.

Table of Contents

List of Figures

List of Tables

List of Abbreviations

Abbreviation	Meaning
CMP	Chip Multiprocessor
TM	Transactional Memory
STM	Software transactional Memory
HTM	Hardware Transactional Memory
RTM	Restricted Transactional Memory
HLE	Hardware Lock elision
TSX	Transactionally synchronized extensions
VLSI	Very-large-scale integration
ILP	Instruction Level Parallelism
SMA	Shared Memory Architecture
WAW	Write after Write dependency
WAR	Write after Read dependency
RAW	Read after Write dependency
SMC	Synchronization control mechanism
TL2	Transactional Locking 2
GVC	Global-versioning counter
RV	Read version
CU	Computational unit
LR	Linear Regression
STAMP	Stanford Transactional Applications for Multi-Processing
API	Application Program Interface (APIs – plural)
RAPL	Runtime Average Power Limit
TX#	Transactional number (e.g. TX1)

Chapter 1

Introduction

Over the past several decades, the performance of general-purpose processors has increased rapidly. This rapid improvement has come both from advances in the technology used to build processor chips and also innovations in architecture of processors. Over the years, improvements in VLSI technology led to smaller and faster transistors and this helped computer architects to increase clock frequency of processors. Furthermore, the number of transistors integrated on a single die is expected to grow according to Moore's law [11] for the foreseeable future. This provides an ample opportunity for processor designers to incorporate more resources in architectural level and boost performance of processors.

The conventional way of processor design was single core processor in which all hardware resources were dedicated to a single processing core. Each generation of processor had larger and more sophisticated components such as caches and reorder buffers. However, by 2005 the performance of single-core processors started to slowdown in computation performance due to "3 Walls": Power Wall, Memory Wall and instruction level parallelism (ILP) Wall [8].

As the single-core processor became more complex, certain limitations made it technologically impossible to achieve better performance. The power wall limitation is met due to increased clock frequency which results in significant heat dissipation. This means that the single-core processor has reached the practical power limit in commodity microprocessors. As for the memory wall, the limitation exists in the gap between the processor and the memory speeds. This gap is increasing over time, requiring the cache sizes inside the processor to be larger in-order to mask the latency of memory. The third wall is related to the dependency of instructions. Single-core processors search stream of sequential instructions and execute independent instructions in parallel. However, the amount of independent instructions found in sequential programs is limited, causing the third wall: ILP wall. The 3 walls together ultimately led to the rise of chip multiprocessors (CMP).

The architecture of a CMP consists of having 2 or more processors integrated onto a single circuit die. This overcomes the limitations of the power wall, memory wall and IPL wall. For Power wall, CMPs are energy efficient and silicon-area efficient due to smaller and less complex cores incorporated into a single chip. For Memory-wall, the computations amongst the cores are overlapped with memory accesses, resulting in better performance. For ILP-wall, there is an increased performance throughput by exploiting parallelism between the cores. Due to these several advantages, the CMP architecture has been the choice of semiconductor manufacturers.

For the last few years, CMPs have taken over the industry by storm. In our present day, CMPs are becoming a necessity in all of our everyday electronics. The cheapest PC/laptop in the market today all consist of at least a dual-core processor. Smart-phones nowadays all have dual-core, quad-core or even octa-core processors. Multi-core processors do not stop there. New cars of today are equipped with multicore systems due to the excessive amount of technologies such as adaptive cruise control, lane departure assistance, self-parking, etc. In present day, new CMPs have transistors of 14nm wide, and the industry is now hitting physical limits. Circuits are now so small that escaping heat is a major problem. While Moore's law may survive another few processor generations, chip manufacturers are starting to change their views on frequency scaling and applying it to core-scaling. This means that instead of focusing on increasing the clock frequency to increase performance of processors, it is now necessary to apply the concept of parallel programming and utilize computational power of multiple cores to boost performance. By utilizing all processing cores of CMPs, it is possible to achieve further performance gain in applications.

1.1 Parallel Programming/Computing

In general terms, parallel programming is the simultaneous use of cores to execute a computational application. Figure 1.1 displays a parallel program consisting of four threads.

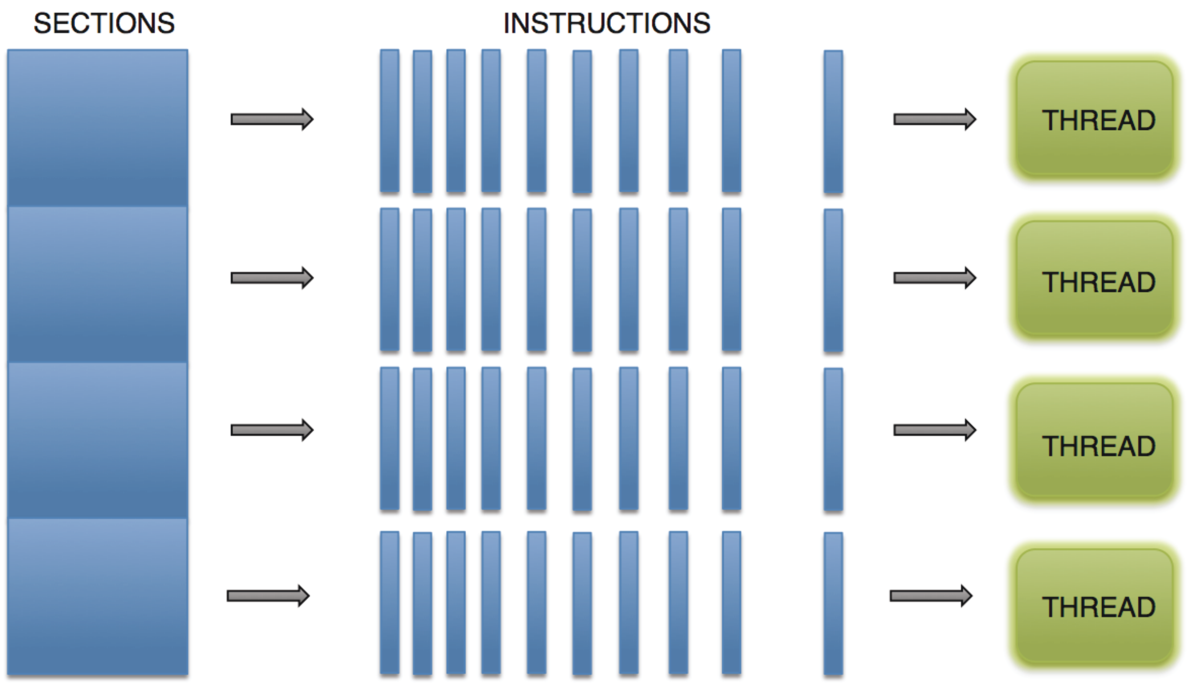


Figure 1.1: Block diagram of a parallel program with four threads

First, the application is broken down into sections that can be executed in parallel (concurrent). Second, each section is broken down further into a series of instructions. Third, these instructions from each part execute concurrently on different threads. Although, this procedure may look simple, it actually consists of a complex order of steps in order to successfully exploiting parallelism in an application. However, there are certain problems that a programmer may face when developing parallel programs. Paul E. McKenny [32] discusses 4 categories that a programmer must take into account while developing parallel programs.

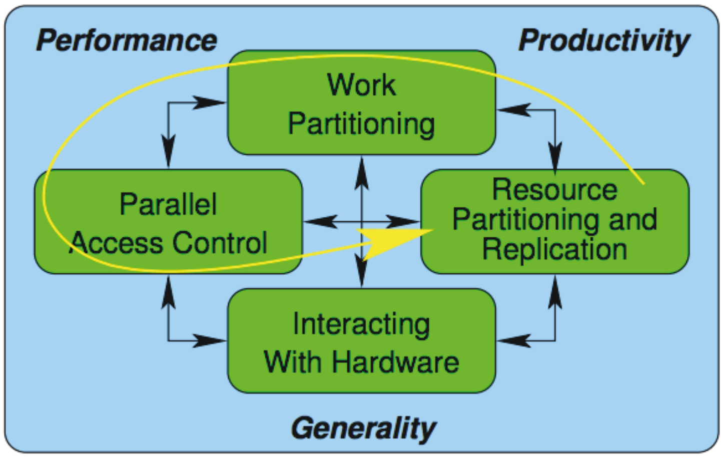


Figure 1.2: Ordering of Parallel-Programming Tasks [32]

1) **Work Partitioning** – is the task of splitting the code or algorithm into discrete sections that can be distributed to be run in parallel across all threads.

2) **Resource Partitioning** – this ensures that the required resources are partitioned for the parallel tasks.

3) **Hardware Interactions** – identifying the resources associated with parallel tasks, such as the operating system, the compiler, number of cores/threads, and other software infrastructures

4) **Control of Parallel Accesses** – is the task of avoiding conflicts such as race conditions on shared memory resources. The programmer needs to synchronize the sequence of the parallel tasks, and often requires serialization (locks) for certain parts of the program. The programmer must also take into account of data dependencies where the order of executions can affect the final results of the program. In shared memory, data dependence occurs from multiple use of the same-shared location accessed by different threads/cores.

Due to these steps and constraints, parallel programming has known to be difficult in applying, or in other terms it is very difficult to get a sequential program and making it parallel.

1.2 Shared Memory Architecture (SMA)

For this thesis, the focus was on Shared Memory Architecture [20] as this is the architecture used in CMPs. SMA is a platform where all threads within a program/application work in a shared space meaning that the memory address space is shared between the threads. In contrast, Distributed Memory (DM) is a method where all threads working in parallel do not share a unified memory address space. Instead, DM uses private memory space for each thread and must communicate with each other explicitly [18].

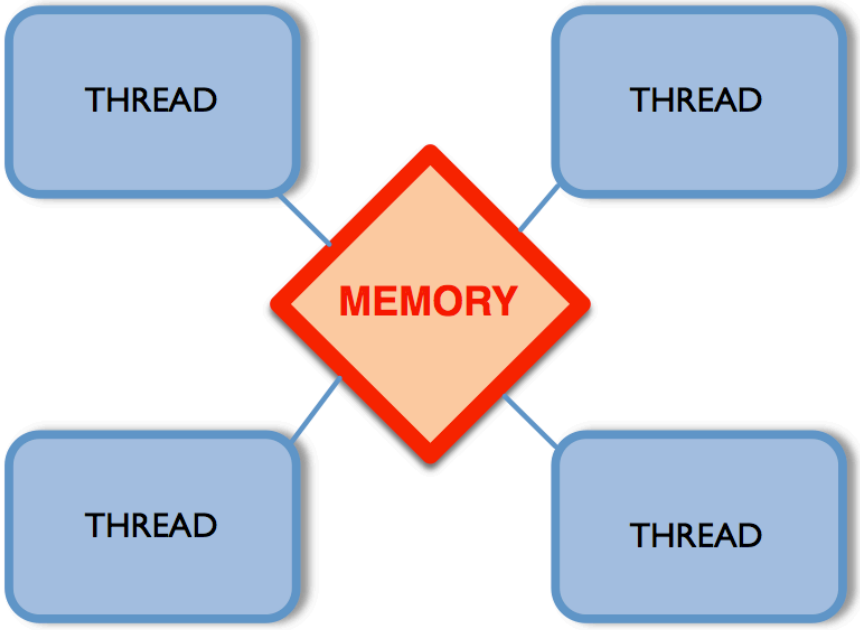


Figure 1.3: Block diagram of Shared Memory

With shared memory, there are some constraints in which a programmer must take into consideration. In SM, threads execute independently but they share the same memory address. It is necessary to have synchronization between the threads that are reading from and writing to SM. This is mainly due to the constraint of only one thread can access the shared memory locations at a time.

SM's major advantage is fast and efficient data sharing amongst the threads as all threads can communicate through a shared memory. One of the major disadvantages of SM is limitation of memory bandwidth where an increased number of threads will require a higher memory bandwidth or else it will cause a bottleneck in performance. Another disadvantage of SM is that it is very prone to data races in which the programmer is responsible for correct synchronization using locks, mutex, semaphores, etc.

1.3 Lock based Synchronization

With shared-memory, there is a high probability that race occurs in programs. This happens when two or more threads are accessing the same address in shared memory. These data races can be classified as dependences: read-after-write (RAW), write-after-write (WAW) or write-after-read (WAR). To avoid these types of data races, a synchronization control mechanism (SCM) must be used. There are many SCMs that can be implemented such as locks, mutexes and semaphores. Locks are the most frequently used SMC in parallel programming. Locks allow a single thread to lock a variable which initiates ownership of a specific shared variable. Once the thread has completed its operation on that shared variable, it unlocks the variable allowing other threads to access the variable. If a lock is being held, other threads cannot access or attempt to acquire the same lock and must wait until it becomes unlocked. There are two types of lock structures that are commonly used: Fine grained locking and Coarse-grained locking.

Fine-grained Locking is used to achieve greater parallelism which leads to better performance. Each fine-grained lock will lock a single shared variable (or very few). Instead of holding a lock for a long time, each thread will hold the lock for a small amount of time while providing protection. Even though fine-grained locking achieves better performance, it has its own drawbacks. Firstly, parallel programming using fine-grained locking is complicated for average programmers. Another major disadvantage of fine-grained locking is high overhead due to the amount of traffic activity taking place with many locks being locked and unlocked.

On the other side, coarse-grained locking is used to lock an entire section of a code instead of a single shared variable. This allows programmers to write correct parallel programs with less complexity because there is only one lock to deal with which means there is less chance of synchronization error. The drawback of coarse-grained locking is less parallelism (low concurrency), which in return leads to low performance. Figure 1.3 shows the general depiction of performance vs. ease of programmability between fine-grained locking and coarse-grained locking.

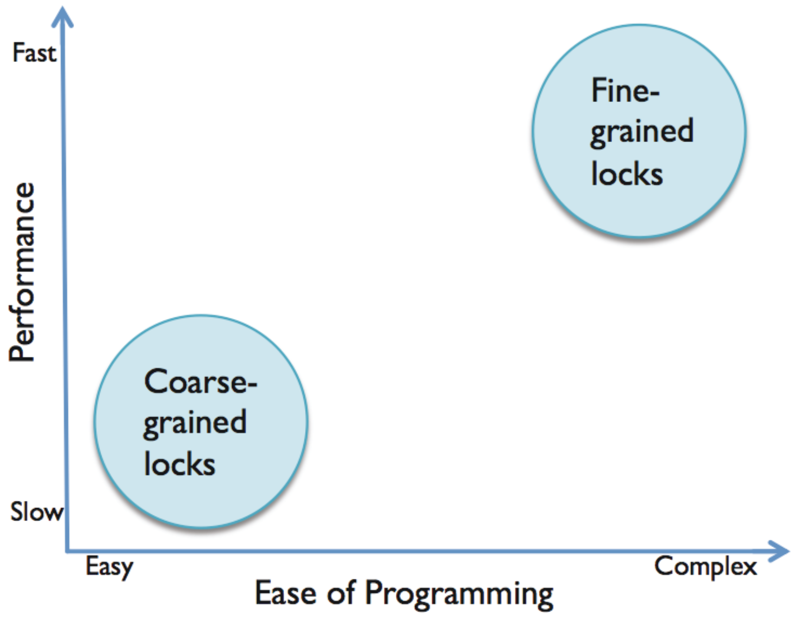


Figure 1.4: Programmability analogy of lock mechanisms

The main challenge in lock-based programming (in particular fine-grained) is tricky synchronization bugs such as deadlock, live-lock and priority inversion. Deadlock occurs when multiple threads stall/wait for each other to release the locks corresponding to the shared variables. This results in a stall, as there is no possibility of forward progress until the lock has been released. For example, thread A holds a lock on resource X and is waiting for resource Y. While thread B holds a lock on resource Y and is waiting for resource X. Both thread A and thread B are waiting and neither of them can proceed.

Live-lock is similar to deadlock as the threads are unable to make forward progress. In deadlock the threads are blocked while in live-lock the threads are not blocked, rather they are busy responding to each other. Priority inversion takes place when a high priority process is blocked (waiting) while a low priority process is executed. Due to these circumstances, this system can become unbalanced and eventually crash. Fine-grained lock-based synchronizing mechanism does promote performance gains but the constraints caused by complex programmability and synchronization bugs prevent it from becoming mainstream.

1.4 Transactional Memory

Transactional processing is not a new discovery; it has been around since the early 1960's known as transactional processing system (TPS). The first TPS was used on American Airlines SABRE computing system, which automated the way the airlines booked reservations for flights [13]. The main idea of TPS was to provide a database of transactions that followed ACID properties:

ACID - Atomicity, Consistency, Isolation and Durability [12]

Atomicity – Each Transaction is atomic which means that if any part of the transaction fails then the entire transaction fails while the state of the system is unchanged.

Consistency – It is necessary in TM where the memory must remain in a consistent state while a transaction is executing. In the case a transaction exits in an inconsistent state, then the transaction is not allowed to complete and will be aborted.

Isolation – Other transactions cannot access data that has been changed by a transaction currently in progress. Isolation is necessary in order to avoid invalid results during execution of a transaction.

Durability – Once a transaction has successfully committed, it cannot be lost in the event of a system crash.

This led to the discovery of Transactional Memory (TM). TM is a parallel programming model, which achieves comparable performance to fine-grained locking while providing ease of programmability of coarse-grained locking [27]. With TM, a programmer only specifies the critical sections of the code to run atomically, while the underlying system will take care of correct execution of the program, reducing the complexity of parallel programming. Transactional memory consists primarily of two types: software transactional memory and hardware transactional memory. In present day, there has been countless amount of research done in this field, due to the fascinating amount of potential it consists of.

1.4.1 Software Transactional Memory (STM)

In software transactional memory, transactions are strictly implemented in software. Shavit and Touitou [30] introduced the first implementation of software transactional memory. STM works by providing a programming model where code is executed in a series of read-sets and write-sets in shared memory. While these reads and writes are being executed their intermediate state is not visible to other transactions. This decreases the probability of conflicts as the window in which transactions execute simultaneously is reduced.

Since the mid-2000, the research in STM has evolved with numerous amounts of concepts and optimizations. These concepts were introduced to further enhance performance of STM systems and also to enhance the ease of programmability. For example in STM, programmers no longer have to handle the case where a transaction aborts. The underlying system of STM will guarantee that the system would eventually commit every transaction by retrying and executing aborted transactions. In present day, there is still ongoing research on STM which shows that there is still potential for further improvements on practical implementations.

There are numerous implementations of STMs. Among those, two are more popular than the rest. The first implementation is Transactional Locking II (TL2) by Nir Shavit et al. [6]. The second implementation of software transaction memory is TinySTM by Pascal Felber et al [31]. TinySTM follows the same structure as TL2 but with enhanced design strategies that achieve even greater performance. Further analysis of TL2 and TinySTM is found in Chapter 2.

1.4.2 Hardware Transactional Memory (HTM)

Hardware transactional memory is the concept of executing transactions in hardware. The primary advantage of HTM is low overhead since it only relies on hardware resources. Recently, HTM has become largely available in commodity processors. Although these implementations have always been best effort meaning that there is no guarantees for forward progress. Some examples of HTM supported by commodity processors include, AMD's advanced Synchronization Series [5], IBM's Blue Gene/Q [1], and Sun's ROCK processor [33]. The recent release of Haswell processor with Intel's TSX (Transactionally Synchronized Extensions) results the widespread availability of HTM on the mass consumer market.

In this thesis, the focus has been on Intel's implementation of HTM called Restricted Transaction Memory (RTM) [15]. Further analysis of RTM is found in Chapter 2.

1.5 Motivation and Purpose

Both STM and RTM have benefits and limitations that either improve or penalize 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 this reduces timing overhead and 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. [30]). This greatly hampers the overall performance in STM systems. Another important difference between 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 (not scalable). 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 constraint and the underlying system deals with majority of the complex synchronization issues, leading to less transactional aborts and a better overall performance in some cases when compared to

RTM.

In this thesis, the focus is on implementing an adaptive system that exploits both STM and HTM 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 [22] to predict whether HTM or STM is faster for a given transaction. The decision tree receives input parameters (such as transaction size, transaction 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 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 [2], NAS [4], and DiscoPoP [37] benchmark suites reveal that on average, the adaptive system is able to improve speed of transactional applications by 20.82%.

1.6 Thesis Outline

The rest of the thesis is organized as follows. Chapter 2 reviews background information as well as research studies relating to TM. Chapter 3 explains design of the proposed adaptive system. Chapter 4 presents the experimental work including methodology and results. Finally, Chapter 5 concludes the thesis and discusses future work.

CHAPTER 2

Background and Related Work

This chapter reviews background information on existing STM systems, Intel's restricted transaction memory (RTM) and the decision tree prediction module. This chapter also examines related literature work focusing on optimization techniques for both HTM and STM.

2.1 Software Transactional Memory

In this section, we explain two popular implementations of software transactional memory. The first implementation is Transactional Locking II (TL2) by Nir Shavit et. al [6]. The second implementation of software transactional memory is TinySTM by Pascal Felber et. al [30].

2.1.1 Transactional Locking II [6]

TL2 is a state-of-the-art word-based Software transaction memory system that uses notion of time to impose order among transactions and guarantee consistency of transitional data. The main feature of TL2 is the ability to handle read and write operations in separate fashion. In TL2, the read operations are invisible; this means that when a transaction reads a shared variable, it will not indicate other transactions that a read operation is taking place. For write operations, TL2 postpones the update to the commit time. This means that TL2 does not perform the update as soon as it executes a transactional write operation; instead, the write operation updates are logged into a local list. Once the transaction is ready to commit, the operation will attain the instruction from the local list. Performance of a STM system is sensitive to the write operations as write operations are the major source of conflicts. By deferring the write operation to the commit time, TL2 reduces the total amount of transactional conflicts in an application. TL2 also utilizes conventional locks and a global-versioning counter (GVC) to validate transactional data. A lock is associated with each shared variable. When a transaction attempts to commit, it obtains the lock corresponding to the variable. GVC is a global counter and is used as timestamp for shared variables. When a transaction starts it copies the current value of GVC into a local variable called read version (rv). The transaction uses rv to validate transactional reads. When a transaction commits it performs an increment-and-fetch on GVC and uses the new value of GVC to tag lock entries corresponding to transactional writes. TL2 is proven to have similar performance to fine-grained locking [6].

2.1.2 TinySTM

The second implementation of software transaction memory is TinySTM. This thesis integrates TinySTM's open source implementation of STM and incorporates it for switching between hardware and software transactions. TinySTM was chosen because it is currently the best performing STM system [31].

TinySTM shares many similarities with TL2. It is also a word-based STM implementation that uses conventional locks to protect the shared memory locations from simultaneous accesses. TinySTM uses the same time-based implementation as TL2, which guarantees transactional consistency. On the contrary, TinySTM contains a different design strategy that differentiates itself from the other STM implementations.

TinySTM uses encounter-time locking which is beneficial for detecting conflicts earlier (increasing transaction throughput). When compared to commit-time locking, conflicts that are detected during commit phase cannot be solved without at least one transaction being aborted. Also, encounter-time locking allows efficient handling of read and write operations without requiring complex mechanisms. For transactional write operations, TinySTM implements two new strategies: Write-through and Write-back. For write-through policy, a transaction writes directly to memory and keeps the old values in a log to reverse updates in the case of an abort. For Write-back policy, a transaction updates memory in the commit phase. TinySTM also provides memory-management functions, which allow transactions to use dynamic memory. This allows the ability to keep track of memory that has been freed (not disposed until commit) or allocated (not disposed until abort). From these design tweaks, TinySTM has become one of the most efficient implementations of software transaction memory [31]

2.2 Hardware Transactional Memory

Hardware transactional memory is the concept of speculative transactions being executed using hardware resources. The primary advantage of HTM is low overhead, since it only uses hardware resources such as level 1 cache, level 2 cache, etc. Recently, HTM has become broadly available in commodity processors. Some examples of HTM supported commodity processors include, AMD's advanced Synchronization Series [5], IBM's Blue Gene/Q [1], and Sun's ROCK processor [33]. Amongst these implementations, the recent release of Haswell processor with Intel's TSX (Transactionally Synchronized Extensions) [15] results the widespread availability of HTM on the mass consumer market.

HTM implementations have always been best effort meaning that they do not provide forward progress. In other words, there is no guarantee that a transaction will successfully commit in hardware; 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.

In 2013, Intel released the first commercially available chip-multiprocessor with HTM support, named Haswell [15]. Along with it, Intel released TSX (Transactionally Synchronized Extensions) to their processor's instruction set. These extensions provide two software interfaces Hardware Lock Elision and Restricted Transaction Memory. [b.6].

- Hardware Lock Elision: a legacy compatible instruction set that provides instructions to lock/unlock shared variables using hardware resources.
- Restricted Transaction Memory: A new instruction set interface, where a programmer identifies a region of code to be executed transactionally. RTM provides no forward progress. Therefore, a program must always provide fallback code to handle a transactional abort that can either restart a transaction or take a non-transactional path (such as locks).

2.2.1 Restricted Transaction Memory

For this thesis, the focus is on Intel's Restricted Transactional Memory (RTM). The proposed Adaptive system uses RTM's intrinsics along with TinySTM, which is used to switch between the two systems. The programming model of Intel's RTM is fairly straightforward to use. In RTM, a transaction is commenced with the instruction XBEGIN. Inside of the transaction, read-sets and write-sets are constructed while other computation operations (branching, arithmetic operations, etc.) can also be executed inside of a transaction. The consistency of read and write sets are maintained in the granularity of cache lines.

If a transaction's read-set/write-set is modified by another transaction, then conflict occurs. In the event of conflict, all the transactions are aborted and only one can proceed. In RTM, a fallback path is needed to guarantee forward process in order to avoid the application or program to stall. To initiate the end of a transaction, the instruction XEND is used. The XEND instruction commits any changes to the shared memory and thus successfully executes the transaction in RTM. RTM provides four transactional instructions:

• XBEGIN initiates the start of a transaction.

• XEND completes a transaction and successfully commits changes to memory

• XABORT aborts the current transaction using an explicit failure code.

• XTEST determines if it is executing within a transaction or not.

```
while(1){                            //loop
int status = XBEGIN;                 //set status bit and start Txn
if(status == _XBEGIN_STARTED){       //status == _XBEGIN_STARTED
    (*g)++;                          //increment shared global variable
    XEND;                    //end transaction
    break;                           //break on success
    }
else{
    ….                               //software fallback code is executed on Txn abort
    }
}
```

Figure 2.1: RTM Pseudo code example

Figure 2.1 shows a sample RTM pseudo code sequence in which all the important instructions are implemented. Inside of the RTM header file, it contains the intrinsics that are used to enable hardware transactional execution. Line 1 starts with a while loop. Inside the loop, there is a status variable that is equal to *XBEGIN.* In line 3, there is an 'if statement' to check if the *status variable == xbegin_started*. if this is true, then the transaction is initiated. Inside the transaction (line 4), there is a shared global variable that is incremented. In line 5, the instruction *XEND* is used to end the transaction. In line 8, in the case of a transactional abort, a fallback path is necessary since RTM does not guarantee forward progress (further information can be found in Section 2.2.3).

2.2.2 RTM Conflict detection and EAX register bits

RTM uses the CPU caches (L1 cache) to track read-sets and write-sets. The conflict detection is handled through the existing cache coherence protocol of the chip multiprocessors. RTM uses eager conflict detection as it keeps transactions in a consistent state by detecting conflicts when a read/write operation to memory has been performed. In RTM, transaction aborts are flagged in the EAX register. 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 a flag is sent to the EAX register with an abort code. Table 2.1 states the abort codes with brief explanation.

Table 2.1: RTM Abort status using EAX abort codes

EAX Status Bit	ABORT	Description
0	XABORT_EXPLICIT	Explicit instruction to abort transaction
1	XABORT_RETRY	Transaction is likely to succeed if retried
2	XABORT_CONFLICT	Interference from another TX
3	XABORT_CAPACITY	Overflow of cache and hardware buffers
4	XABORT_DEBUG	Debug breakpoint
5	XABORT_NESTED	Transaction aborted within nested transaction

Conflict and capacity aborts take up the majority of transactional aborts in RTM. Conflict aborts occur when a transaction interferes with concurrent memory operations (read/write) performed by another transactions. Once this abort is triggered, the processor will abort the transaction by discarding all the updates done to the shared memory. Capacity aborts occur where there is an overflow of buffers and the capacity of the cache has been reached which results in an automatic transaction abort.

2.2.3 Cache Coherency Conflict Detection

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 same 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 [17] 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. RTM follows the MESI protocol for cache coherency.

2.2.3.1 MESI protocol [34]

MESI is a type of invalidation-based protocol, which supports write-back caches. MESI is the acronym for the four states that each cache line can transition to:

- Invalid – This is considered the non-valid state. This means that the data is not located in the cache or the local copy of the data is incorrect due to another process updating the memory.
- Shared – This state is used for those cache blocks that are not changed by any processor.
- Exclusive – The state is exclusive when a cache is the only one that has the correct value of the block.
- Modified – This state is used for those cache blocks that are written by processors..

Figure 2.2 depicts MESI state diagram. The following is a brief explanation of how the MESI protocol works. The initial state of a cache block is invalid. When a processor writes to a cache block for the first time, the state changes to modified as there are no copies of the block in other caches. If a processor reads a block for the first time, it broadcasts BusRd command on the interconnection network. The cache that has the block sends it to the requester. Also, the state of the block changes to shared in both requester and the sender as more than one cache hold the data. If processor reads a block and no other cache has the block, then the memory provides the corresponding data and state changes to exclusive.

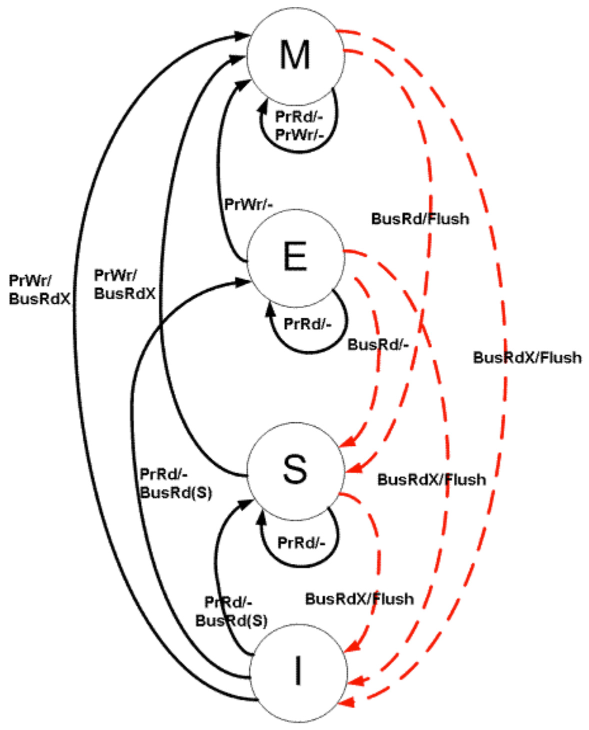


Figure 2.2: State diagram of MESI protocol [34]

2.2.4 RTM Restrictions and Limitations

Intel's Restricted Transactional Memory has the term 'restricted' because it is very prone to transactional conflicts, which are primarily due to both hardware and software operations. There are many operations in RTM that are labeled as restricted and if a restricted operation is attempted then the transaction is aborted and the fallback path is executed.

These are some restrictions in RTM:

- Debugging tools are not supported meaning that if any breakpoints are located inside of a transaction, it will be automatically aborted.
- Interrupts located inside of a transaction will cause an abort before the calling of the interrupt handler
- Input/output statements will cause an automatic transaction abort. For example, a ‘printf’ statement will causes RTM to abort.
- Software/System operations such as context switching and page faults cause transactional aborts.
- Hardware resources that exceed the capacity of the cache will cause a transaction abort. If a transaction’s read-set or write-set does not fit in the CPU cache, it will result in a transaction abort due to the overflow of the internal processor buffers.
    - Cache size in Haswell is 32KB with 8-way associativity.
- Unnecessary aborts due to false sharing of cache lines.
    - If two transactions share a cache line and one of them aborts due to conflict over a shared variable in the cache line, the other transaction is aborted too.

2.2.5 RTM’s Fallback Path

Commodity chip multiprocessors (such as Haswell processor) that support hardware transactional memory, use the ‘best-effort’ mechanism. This basically means that there is no guarantee for a transaction to succeed even if there is no conflict. In RTM, a fallback policy is necessary to provide forward progress. A fallback policy is typically executed after the threshold of RTM’s retry count has been met. The retry count is the number of times an aborted transaction retries execution. This is important since transactions in RTM have an abundant reasons to abort (refer to section 2.2.4). 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. Further information on fallback path can be found in Section 3.3.1.

2.3 Decision Tree

For this thesis, the focus is on switching between hardware transactional memory and software transactional memory at transaction granularity. By using Decision Tree C4.5 [23], we are able to predict which system is the best choice for a given transaction.

Decision tree uses groups of input datasets and generates a tree as output that resembles a tree diagram where each branch is a decision. Ross Quinlan developed the early stages of the Decision Tree in 1979 (ID3 algorithm) [22]. In 1993, the C4.5 algorithm was developed to increase accuracy of Decision Tree. The C4.5 builds decision trees from a set of training dataset using information entropy. The decision tree consists of three nodes which are root, branch and leaf. At each branch of a tree, the C4.5 algorithm attains the attribute of the data that effectively splits the set of samples into sub-group in each specific class. This splitting process is referred to as information gain (differences in entropy). The input dataset contains the parameters of a function. In this thesis, the focus is targeted on transactional parameters such as transaction size, read-set size, etc. The output of the decision tree results in a binary value of 0 or 1, which represents the predicted outcome. For this thesis, the outcome of the decision tree represents whether RTM or STM will be used to execute a transaction.

2.4 Related Work

Irina Calciu et al. [14] presented Invyswell, a hybrid transactional memory system that incorporates RTM and InvalSTM. InvalSTM is a modified STM system that was created [21] previously. One of the key differences between InvalSTM and other STMs is that it performs commit-time invalidation. This approach identifies conflicts with other concurrently executing transactions during its commit-phase. InvalSTM also implements bloom filters for conflict detection between HTM and STM. For Invyswell, each transaction is first tried in hardware. If the hardware abort status suggests that a transaction is unlikely to succeed in hardware, then it is retried in InvalSTM. They also investigate RTM's limitations and restrictions and provide InvalSTM as a fallback policy instead of using lock mechanism. They also incorporate optimizations such as *failfast*. This optimization 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 our study, we do not use STM as a fallback policy for RTM; instead, we implement independent switching between RTM and STM. Also, our adaptive system is static and its runtime overhead is low. Furthermore, Invyswell is not evaluated from energy point of view. On the other side, we examine energy efficiency of our adaptive system and compare it with both HTM and STM.

M. Wang et al. [29] exploited Intel's restricted transaction memory to implement a molecular dynamics simulator called Moldyn. They explore several important relationships between transaction size and write ratio inside transactions as well as retry count and transaction abort rate. They investigate how these parameters affect the overall performance of an application. They introduce code transformations such as computation splitting and privatization for improving performance. Computation splitting/merging is the basis of transactional aborts caused by the size of a transaction, which can lead to low performance. In this paper, they identify a 'sweet-spot' in the Moldyn application where they compute each pair of molecule updates inside a single transaction as opposed to thousands of molecules or single molecule in a transaction. This 'sweet-spot' in transaction size increases performance in RTM.

For this thesis, we incorporated this paper's notion of the correlation between the transaction's characteristics and the performance impact. We exploited the parameters of a transaction such as transaction size, read-set size, write-set size, etc. and provided a TM system based on both HTM and STM. This is important because these parameters give information on a system's behavior and constraints. By using these factors, we are able to switch between HTM and STM at transaction granularity to achieve performance gain.

Pereira et al. [28] presented an extensive evaluation of Haswell's Transaction Memory performance. They focused on RTM's forward-progress polices since Intel's TSX does not guarantee that a transactional execution will commit. This technique retries the execution of a transaction with or without a time delay and attempts completing the transaction execution speculatively. They introduced three policies for forward progress: Maximum retry, Back-off and SerControl. Maximum retry is the simplest approach as it limits the number of times a transaction can be retried. Once a transaction reached the retry threshold, it will commence the fallback policy with a global lock. Back-off policy is based on a time delay in which an aborted transaction will wait for a time delay before restarting. The duration of this time delay is uniform as the time delay increases exponentially for every restart. Once again, there is a threshold for number of transactional retries and once it has reached the limit, the transaction will be executed using global lock. The next policy that was introduced in this paper is SerControl. This policy focuses 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 maximum retry policy is applied. There are also other aborts that are considered such as page-fault that may occur again if the transaction is immediately retried so the back-off policy is applied after the threshold has been reached. It is important to note that this paper focuses on increasing the probability of executing transactions successfully in RTM.

In our thesis, we incorporate the ideas of the potential performance benefits of forward progress policies. Although, the notion of having an efficient forward progress policy is important, the actual performance gains are negligible. In this research paper [28], they do not show the comparisons between the proposed RTM forward progress policy and another TM system such as TinySTM. This would have clearly indicated the impact of this paper's proposed policies on performance. For our study, we conducted many experimental tests with a variety of retry counts for transactional aborts. 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 at 4. Based on 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. Furthermore, in our work, we investigated the behavior of a transaction that best suits 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 parameters of a transaction.

M. Castro et al. [26] presented a dynamic approach to do efficient thread mapping using machine learning. This technique relies on matching the behavior of an application with the system characteristics. This technique is a dynamic approach and gathers information from the application and the STM system at specific time intervals. They compared dynamic approach with static thread mapping approach based on machine learning. For the Static approach, they used the decision tree learning method which was trained using datasets of input parameters. It will then output a decision tree that will predict a thread mapping strategy. The predictor chooses one of four different strategies: round-robin, scatter, compact and Linux. For dynamic Thread mapping, there are three phases: hardware topology analyzer (HTA), thread mapping predictor and transaction profiler. The HTA uses hardware locality library to get information from the underlying platform topology (hierarchy of caches and how they are shared among caches). The transaction profiler gets information from hardware counter and from the TM underlying system all during runtime at specific time intervals. The thread mapping predictor gets the data from the profiler and feed the data to a decision. Then, the predicted thread mapping strategy is applied. Whenever a TM operation starts, aborts or commits, the transaction profiler will be executed during these intervals and calls the thread mapping predictor to switch strategies when necessary. For the transaction profiler, only one concurrent running thread will be chosen for that task because it reduces stress on the system and there isn't any need for extra synchronization mechanisms for all threads. The experimental results shows that thread mapping strategies do have a major impact on performance. Out of the 56 TM applications, only 3 applications show no performance gain and 8 applications had performance loss. The maximum performance loss was 8% due to wrong predictions of the decision tree.

In our thesis, 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 in this thesis, the focus was on the C4.5 algorithm. C4.5 is an enhanced version of ID3, as it also supports continuous attribute that results 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 consists of low, medium and large transaction sizes as well as low, medium and large working set size.

C. Wang et al. [3] presented optimizations for limiting overhead in software transaction memory. They focus on supporting transactional code in unmanaged languages such as C. Optimization of STM overhead in unmanaged languages is a challenging task as it requires implementing validation in the granularity of the cache block rather than an object. In this paper, they proposed techniques to allow programmers to initiate blocks to be executed atomically. They also exploit compiler-based optimization techniques such as in-lining (necessary for fast paths), eliminating redundant barriers and register checks. Our work is orthogonal and can be combined with this paper [3] to enhance performance further.

Z. Li et al. [37] presented a compiler-based tool, called DiscoPoP, to automatically identify regions of code that can be executed in parallel. It is designed to be able to find code regions with arbitrary granularity. It is important to note that DiscoPoP finds regions of a code in which data dependency does not exist. This is called CU (computational unit). In the next step, dependency graphs are then built. The nodes in the graph represent CUs and the edges represent the dependency between the CUs. By exploiting the dependency graph, DiscoPoP determines the potential parallelism that is available on different levels of the sequential code.

For this thesis, the DiscoPoP parallel benchmark suite was used to evaluate adaptive system. This Benchmark suite consisted of small and medium sized transactions that consisted of medium sized working set. For the decision tree training phase, it is important to have a wide range of transactional parameters to achieve greater accuracy in predictions.

D. Didona et al. [8] presented a self-tuning optimization technique to dynamically adjust the concurrency level in STMs. The purpose of this paper is to automatically identify the optimal degree of parallelism which will maximize the throughput of the applications. They introduced self-tuning methods for both shared-memory and distributed STMs. The performance of a TM application varies based on different factors such as duration of transactions, level of data contention, ratio of update vs. read-only transactions, etc. By changing the number of threads at runtime, it can improve the performance of some applications instead of having a fixed number of threads. In this paper, they used the self-tuning method that combines exploration-based and model-driven approaches. Shared-memory STMs use the exploration-based approach which consisted of three phases. The first phase is measurement phase. In this phase, the application runs with fixed number of threads and measures the number of commits and aborts. The second phase is decision phase. This phase decides whether to increase or decrease the number of threads until the maximum is reached. The third phase is transition phase. This phase is an external controller thread which either adds or removes threads from an application depending on the results from the decision phase. Distributed STM uses an analytical-based performance model which relies on a set of assumptions based on transaction conflict patterns.

For this thesis, a similar approach is taken regarding the evaluation phases that are introduced in this paper. The decision tree consisted of two phases, training and testing. This was done similar to this paper in order to have discrete evaluations based on the decision tree prediction module.

Y. Rughetti et al. [35] proposed a technique which automatically tunes the degree of parallelism in HTM. To achieve automatic tuning, the authors incorporated a machine leaning algorithm. This work focuses on a two-layered approach where the first-layer is the correction functions which is used to predict values of time. The second-layer consists of the performance predictor model that predicts the level of concurrency. There are existing STM approaches such as Hill-climbing techniques [24] and transaction scheduler [9] that optimize degree of parallelism. The hill-climbing technique changes the parallelism degree by reacting to throughput or abort rate. Transaction Scheduling is the basis of mapping transactions to threads dynamically to minimize data contention, and then the rescheduled threads are removed from the execution for that time interval. This approach gives different types of information from abort ratio to the details on a transaction read/write set. In these approaches, the predicted value of the transaction wasted time is used to find the system throughput. This allows predicting the optimal value to achieve the expected maximum throughput. In STM, it is easier to access via software instrumentation to monitor specific parameters. However, these parameters are not supported in HTM, and implementing it in HTM via software would create overheads and lower the performance severely, especially since an advantage of HTM is supposed to avoid any costly additional software instrumentation (overhead). The techniques are not compatible for HTM since all of these models for STM do not take into account the transaction aborts in which HTM is very vulnerable to conflicts. In this study, the authors implement a classification approach comparing two different machine-learning methods: Decision tree and Neural Networks. This approach consists of constructing a training set for a specific application. The training set is constructed by executing a few runs of the application with different inputs of configuration parameters. For each input, the application is executed for a range of threads. By implementing this for each workload tested during the training phase, it becomes possible to determine the best performing concurrency level. The major benefit of this approach is that it follows the one-step layered approach meaning that it does not require the usage of correction functions.

In this thesis, we use machine learning to determine which TM system is appropriate for a transaction. This paper also shows the importance of overheads associated with HTM systems in which careful analysis must be taken or else it will cause performance penalties. Furthermore, our adaptive system does not execute both HTM and STM, simultaneously. As this process incurs extra overhead. Thus, the adaptive system avoids this performance penalty by allowing a transaction to execute in either hardware or software. (Further information can be found in section 3.3)

Y. Xiao et al. [36] proposed an optimization technique that statically decides on transactional parameters to improve performance of STM in parallel applications. By focusing on a transaction's characteristics (such as transaction size, read set size and write set size), it is possible to achieve speedup in applications. The transaction size is a crucial parameter that can have significant impact on performance and it is important to have an optimum size to achieve speedup. If the transaction size is too small, it can lead to overhead that exceeds the performance gain of parallel execution (results in slowdown when compared to sequential programs). If the transaction size is too large, it can lead to an excess amount of rollbacks due to a higher probability of transactional aborts in applications. Thus, it is important to have the optimum 'sweet spot' of transaction size. This approach of optimizing each parameter manually can be a tedious and time consuming process. To overcome this issue, the authors propose two optimization techniques that are designed to automatically determine the optimal transaction size. The first technique exploits Linear Regression (LR) to predict the transaction size. The LR works by attaining the transaction parameters such as transaction size, read-set size, write-set size and predicts the optimum transaction size. However due to the simplicity of implementing LR, the accuracy is quite low. In order to improve the accuracy, multiple LR models are used to predict transaction size. In addition, a decision tree prediction model determines which LR model is appropriate for a given transaction. Overall, these optimization techniques improved the performance of STM based applications.

For this thesis, the adaptive system incorporates both HTM and STM to enhance performance of parallel applications. A decision tree is implemented to predict the optimum system based on a transaction's characteristics. With the optimization techniques proposed by Yang et al. [35], there is an opportunity to enhance the adaptive system by optimizing STM and RTM separately based on the transactional characteristics (such as transaction size, read-set size, write-set size, etc.).

Chapter 3

Adaptive System Design

This chapter describes the design of the proposed Adaptive system. Section 3.1 explains the importance of transaction granularity for the proposed adaptive system. Section 3.2 analyses the programmability aspects of RTM. Section 3.3 revolves around the synchronization technique that is used to seamlessly switch between RTM and TinySTM. Section 3.4 depicts how the adaptive system is implemented into a specific source code. Finally, section 3.5 explains the implementation of the decision tree prediction module.

3.1 Transaction Granularity

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 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, the 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 to execute in either hardware or software, but not both.

In order to achieve transaction granularity, program counter (PC) was used to distinguish each and every transaction. While executing an application/benchmark, we are able to attain the parameters of each transaction. These parameters include elapsed execution time, the static size of transaction, transactional data, the number of aborts, etc. From these parameters, it is possible to further understand the behaviors of both RTM and TinySTM.

```
1.      uint32_t eip1 = 0;
2.      __asm__ __volatile__("movl $., %0" : "=r"(eip1));
```

Figure 3.1: Program counter Code sequence

Figure 3.1 shows the code sequence used to read PC. When a transaction is initiated this code returns address of the first instruction in the transaction. Line 1 initializes *eip1* (a local variable). Line 2 retrieves the value of the program counter to identify each transaction. This is crucial for the adaptive system as it switches TM systems from one transaction to another.

3.2 Restricted Transaction Memory (RTM)

The implementation of RTM programs was based on the programmability references from Intel's TSX manual [17]. The key factors of an RTM program is the following:

- Retry count
- Fallback policy
- Transactional abort status

The retry count is the maximum number of times an aborted transaction is rolled back and retries execution. This is important in RTM since transactions have an abundant reasons to abort (refer to Section 2.2.4). By retrying an aborted transaction 'x' number of times, there is a possibility that the transaction can eventually commit in hardware. Once the maximum retry threshold is reached, the fallback policy is applied. It is important to be able to execute a transaction using hardware resources as often as possible in order to use the performance benefits of RTM. 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 due to wasted work. For example, if the retry count is 12 and the application triggers capacity aborts, the program will keep retrying the execution until the threshold of retires is met. This wastes processor cycles and the outcome is performance slowdown. Also, having a low retry count can cause the fallback policy to be executed too prematurely. This means that RTM does not have a chance to be executed, which is detrimental for performance gain. By conducting experimental test cases, having a retry count of 4 is a 'sweet-spot' for optimal performance.

In RTM, it is necessary to incorporate a fallback policy to guarantee that an application will successfully execute. A sample code sequence of RTM's fallback policy is found in figure 3.2. This code sequence is placed inside a header file (tm.h in STAMP) and is executed when RTM is called upon. The tm.h file contains the APIs necessary for transactional execution for both software and hardware transactions. For this thesis, these APIs are modified to support TinySTM and RTM.

This code sequence in figure 3.2 only focuses on the lock mechanism that is used in the case of an RTM transactional abort. This fallback policy consists of a global pthread lock. In line 4, if the number of tries is less than 0, then the fallback path is initialized by acquiring a lock. This sequence happens during *TM_BEGIN_RTM*. Once the *TM_END_RTM* is called, the code sequence will try to commit a transaction in RTM only if the number of tries is greater than 0. Otherwise, in line 11, the pthread lock that was held previously is released and the transaction is executed using locks. This guarantees forward progress as the transaction will eventually commit after the threshold of retires has been met and eventually executes the transaction using locks.

```
#define TM_BEGIN_RTM()
    ...
    tries --;
    if (tries <= 0)
    pthread_mutex_lock(&global_rtm_mutex);
    ...

#define TM_END_RTM()
        if (tries > 0)
        ...
        else
  pthread_mutex_unlock(&global_rtm_mutex);
        ...
```

Figure 3.2: RTM fallback policy

In RTM, there are many constraints that result in a transactional abort. To track these aborts, RTM uses the EAX status register to specify the exact cause of an abort. Once a transaction aborts, the EAX register will send an abort code with the reason of abort (further information on EAX abort codes can be found in Section 2.2.4).

To measure the cause of aborts, we use an array to keep track of all the different kinds of transactional aborts inside an application. Once the application executes, the total number and type of aborts will be printed out. This feature is an important aspect for understanding the behaviors of RTM. From initial evaluation of RTM, the benchmarks that perform poorly tend to have a higher abort rate with the majority being capacity aborts. While the benchmarks that show performance gain have minimal abort rate, along with minimal capacity aborts. Capacity aborts are detrimental to RTM's performance as the hardware resources are bounded with constraints. A benchmark that consists of a large working set size, and/or large transaction size, has a higher probability of getting capacity aborts in RTM, thus decreasing performance. Figure 3.3 depicts the EAX status bits located in the RTM header file.

```
/* Status bits */
#define XABORT_EXPLICIT_ABORT      (1<<0)
#define XABORT_RETRY               (1<<1)
#define XABORT_CONFLICT            (1<<2)
#define XABORT_CAPACITY            (1<<3)
#define XABORT_DEBUG               (1<<4)
#define XABORT_STATUS(x)           (((x) >> 24) & 0xff)
```

Figure 3.3: EAX status bits found in RTM header file

```
 1.     ...
 2. {
 3. (tx->num_abort)++;
 4.
 5.     if((eax_regg & 0x01) == 0x01)
 6.    (tx->abort_explicit)++;
 7.     if((eax_regg & 0x02) == 0x02)
 8.    (tx->abort_retry)++;
 9.     if((eax_regg & 0x04) == 0x04)
10.    (tx->abort_conflict)++;
11.     if((eax_regg & 0x08) == 0x08)
12.    (tx->abort_capacity)++;
13.     if((eax_regg & 0x10) == 0x10)
14.    (tx->abort_debug)++;
15.     if((eax_regg & 0x20) == 0x20)
16.    (tx->abort_nested)++;
17.  ...
```

Figure 3.4: Implementation of EAX status register

The EAX status bits are implemented in conjunction with RTM's header file that consists of the definitions of the aborts. This code sequence is placed inside RTM_stats function to attain all the metadata of a transaction. In line 3 (figure 3.4), the total number of aborts is accumulated. From line 5 to line 16, there are if statements to check whether EAX status bit are initialized. For example, if there is an abort, it will check each status bit and once the status bit is found, it will determine the cause of abort. These abort metadata is then accumulated in the array structure to attain all the aborts of a transaction within an application.

3.3 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 shared memory by either one of the systems. To enable mutual-exclusion of RTM and STM, we exploit a conditional variable. The pseudo code in Figure 3.5 and 3.6 shows how synchronization is handled between the two systems.

The synchronization occurs inside the functions *tx_start*() and *tx_commit*() which depict the start and commit phases of a transaction, respectively (please refer to figure 3.5 and 3.6). These functions have other code sequences but are taken out in order to only focus on the synchronization portion. 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 *txstart*() 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 transaction (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. The same procedure is followed for software transactions (lines 37-44).

```
tx_start(int rtm_n_stm)
{
  ...
  if(rtm_n_stm == 1)
  {
    pthread_mutex_lock(&rtm_stm_sync_mutex);
    while (num_in_flight_stm > 0)
        pthread_cond_wait(&sync_cond_rtm, &rtm_stm_sync_mutex);
    num_in_flight_rtm++;

    pthread_mutex_unlock(&rtm_stm_sync_mutex);
  }

  if(rtm_n_stm == 0)
  {
    pthread_mutex_lock(&rtm_stm_sync_mutex);
    while (num_in_flight_rtm > 0)
        pthread_cond_wait(&sync_cond_stm, &rtm_stm_sync_mutex);
    num_in_flight_stm++;

    pthread_mutex_unlock(&rtm_stm_sync_mutex);
  }
  ...
}
```

Figure 3.5: Pseudo code for synchronization of RTM and STM in tx_start().

```
tx_commit(int rtm_n_stm)
{
  ...
  if(rtm_n_stm == 1)
  {
    pthread_mutex_lock(&rtm_stm_sync_mutex);
    num_in_flight_rtm--;
    if(num_in_flight_rtm == 0)
    pthread_cond_broadcast(&sync_cond_stm);
    pthread_mutex_unlock(&rtm_stm_sync_mutex);
  }

  if(rtm_n_stm == 0)
  {
    pthread_mutex_lock(&rtm_stm_sync_mutex);
    num_in_flight_stm--;
    if(num_in_flight_stm == 0)
    pthread_cond_broadcast(&sync_cond_rtm);
    pthread_mutex_unlock(&rtm_stm_sync_mutex);
  }
  ...
}
```

Figure 3.6: Pseudo code for synchronization of RTM and STM in tx_commit().

In this synchronization step, there are important lock functions to promote atomicity. In line 8, the instruction *pthread_cond_wait()* is called. If a transaction is being executed in STM mode, then this function blocks the calling transaction. When the last transaction in STM mode commits, it broadcasts a signal (line 42) and wakes up all blocked transaction.

3.4 Implementing Source Code

The main goal of our adaptive system is to have a uniform design of incorporating both systems. Typically, in TM applications/benchmarks, there are macros that enable transactions to begin and end as well as macros for data access such as reads and writes. For our adaptive system, there are new instructions dedicated to RTM and TinySTM. For a given TM application/benchmark, by substituting the source code with RTM and TinySTM macros, our adaptive system is able to seamlessly switch between systems for different transactions. These macros are defined in header files which consist of the entirety of the RTM and TinySTM codes. Figure 3.7 shows sample code of how RTM and TinySTM work alongside each other using the proposed macros.

To start and end a transaction in RTM, we use the macro *TM_BEGIN_RTM* and *TM_END_RTM*. The same structure of macros is used to start and end a transaction in STM: *TM_BEGIN_STM* and *TM_END_STM*. There are two macros for transactional data access in RTM: *TM_SHARED_READ_RTM* and *TM_SHARED_WRITE_RTM*. Similar structure is used for STM: *TM_SHARED_READ_STM* and *TM_SHARED_WRITE_STM*.

```
TM_BEGIN_STM();
TM_SHARED_READ_STM();

        //transactional area...

TM_SHARED_WRITE_STM();
TM_END_STM();


TM_BEGIN_RTM();
TM_SHARED_READ_RTM();

        //transactional area...

TM_SHARED_WRITE_RTM();
TM_END_RTM();
```

Figure 3.7: Pseudo code for implementing RTM and STM.

3.5 Implementation of Decision tree

Decision tree is an effective method of supervised machine learning that exhibits an accurate prediction based on a group of datasets [22]. The goal of implementing a decision tree is to create a model that predicts a value based on a set of input parameters. Our Adaptive system exploits a decision tree prediction module (C4.5 algorithm [23]) to be able to predict which TM system is the better choice for a given transaction. The C4.5 algorithm was chosen because of the stability and good accuracy when compared to other prediction model algorithms [22]. 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 (further information can be found in Section 2.3). This process can be broken down into two phases: training phase and testing phase.

3.5.1 Training Phase

The training phase is conducted to attain a prediction model based on the decision tree. The input datasets are constructed using the following transaction parameters:

- Transaction size
- Read-set size
- Write-set size
- Write-ratio

Transaction size refers to the operations that are present inside a transaction. Typically, a transaction is initialized with *TM_BEGIN* and a transaction is committed with *TM_END*. In between these instructions lie different operations, such as arithmetic operations, read-sets, write-sets, 'for' loops, etc. One way to measure transaction size is counting 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 the transaction size. Since all C codes are compiled to assembly instructions, we use the number of assembly instructions to measure transaction size. In general, transaction size is important when conducting evaluations for RTM. This is primarily due to the hardware resource constraints. Once the cache capacity of RTM has been reached, there is a higher probability of the transaction resulting in an abort, thus resulting in overall performance slowdown.

Another important transactional parameter is the working set size which is defined as the number of distinct memory locations accessed. This includes both the read and write sets inside of a transaction.

Transaction conflict is more likely to occur in applications with large working set size. In RTM, such conflicting accesses force an abort to ensure that atomicity of the transaction is preserved, yet this will result in performance slowdown. The write-ratio is the ratio between the number of shared writes and the total number of shared accesses. The write-ratio is used as another parameter that is included in the training set of the decision tree, in order to improve the accuracy of prediction.

Overall, these parameters are important in terms of the behaviors of both RTM and TinySTM. RTM favors small sized transactions as well as small working set size. While in STM, there is much more flexibility and offers better performance than RTM for large transaction sizes and large working set sizes.

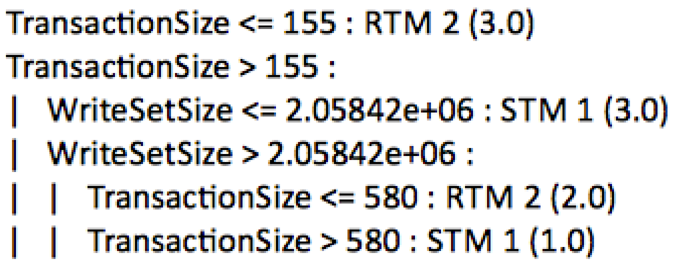
The training phase consists of a set of benchmarks that are chosen based on small, medium and large transaction sizes and working set sizes from all the 3 benchmark suites (STAMP, NAS and DiscoPoP).

The following are the benchmarks used for the training phase:

- GENOME
- LABYRINTH
- YADA
- Embarrassingly Parallel
- Montercarlo_Pie
- Light_Propogation

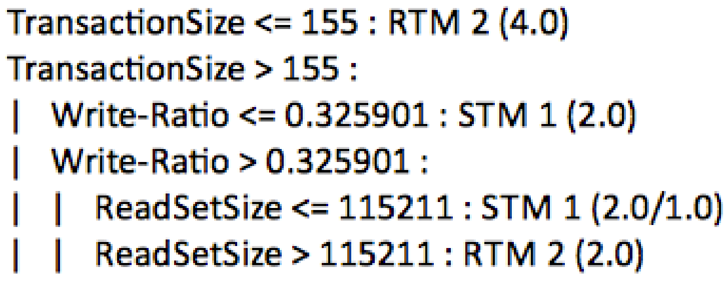
The benchmarks are executed twice: once using RTM and the other using TinySTM. The Decision tree is trained based on statistics generated by RTM and TinySTM. This procedure was done separately for 2, 4 and 8 number of threads because the characteristics of a transaction can vary as the thread count increases. The output of the decision tree is a binary bit that indicates whether RTM or STM is better for a given transaction.

Figures 3.8, 3.9 and 3.10 represent the decision tree predictions for 2-, 4- and 8-thread, respectively. Based on these predictions, an evaluation was conducted on a separate set of testing benchmarks. The result of the decision tree follows an if/else procedure. Figure 3.8 corresponds to the prediction for 2 threads. First, it checks transaction size. If the transaction size is less than 155, then the optimum TM system is predicted to be RTM. Else, if the transaction size is greater than 155, it enters the next set of base parameters to be examined. Now, if the write-set size is less than 2.05×10^6 then the TM system that should be used is STM. Else, if the transaction has a larger write-set size then it will check the next base parameter. Once again, the decision tree checks if the transaction size is less than 580, then RTM will be used; otherwise, STM system will be used. As the thread count increases, the transactional execution time can change and ultimately the predicted system can change. In order to overcome this issue, the decision tree is implemented separately for threads 2, 4 and 8 to improve accuracy.



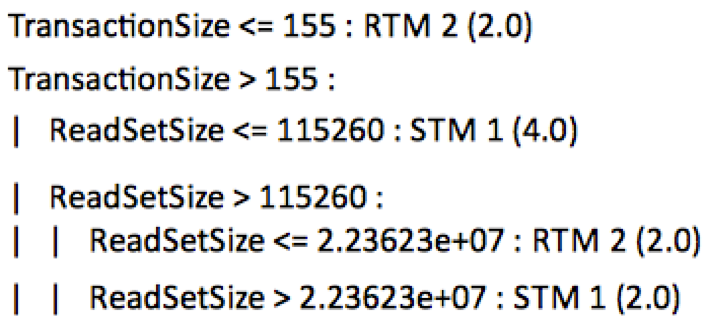
```
TransactionSize <= 155 : RTM 2 (3.0)
TransactionSize > 155 :
|   WriteSetSize <= 2.05842e+06 : STM 1 (3.0)
|   WriteSetSize > 2.05842e+06 :
|   |   TransactionSize <= 580 : RTM 2 (2.0)
|   |   TransactionSize > 580 : STM 1 (1.0)
```

Figure 3.8: Decision tree output for 2 Threads



```
TransactionSize <= 155 : RTM 2 (4.0)
TransactionSize > 155 :
|   Write-Ratio <= 0.325901 : STM 1 (2.0)
|   Write-Ratio > 0.325901 :
|   |   ReadSetSize <= 115211 : STM 1 (2.0/1.0)
|   |   ReadSetSize > 115211 : RTM 2 (2.0)
```

Figure 3.9: Decision tree output for 4 Threads



```
TransactionSize <= 155 : RTM 2 (2.0)
TransactionSize > 155 :
|   ReadSetSize <= 115260 : STM 1 (4.0)
|   ReadSetSize > 115260 :
|   |   ReadSetSize <= 2.23623e+07 : RTM 2 (2.0)
|   |   ReadSetSize > 2.23623e+07 : STM 1 (2.0)
```

Figure 3.10: Decision tree output for 8 Threads

Table 3.1 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.

Table 3.1: 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

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 transaction sizes that range 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.

3.5.2 Testing Phase

The testing phase is conducted to predict whether RTM or STM is better for a given transaction. The testing phase consists of 6 different benchmarks, which are:

- Conjugate-Gradient
- Multi-Grid
- KMEANS
- SSCA2
- Ann_Training
- Mandelbrot

The reason why there was no inclusion of the training benchmarks for evaluation is due to having a 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).

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 groups of 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 the execution time of transactions. These parameters vary from one benchmark to another. Table 3.2 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.

Table 3.2: Decision tree prediction based on Transaction Granularity for Benchmark CG

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 made 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.

In summary, the primary goal of the adaptive system is improve performance of parallel applications by incorporating the notion of switching between hardware and software transactions within a given application. A decision tree is incorporated to predict the optimum system for each transaction based on its characteristics.

Chapter 4

Experimental Results

The motivation to develop the proposed adaptive system is originated from the benefits and limitations of both TM systems. Depending on an application's transaction characteristics, either RTM or STM can outperform each other. This chapter focuses on the experimental analysis of the adaptive system based on the testing benchmarks. In Section 4.1, we explain experimental framework and benchmark specifications used to evaluate the adaptive system. Section 4.2 analyzes both RTM and TinySTM on the basis of performance and energy-delay. Section 4.3 reports performance and energy-delay of both RTM and TinySTM.

4.1 Experimental Framework and Benchmark Specifications

In this thesis, the focus is on simulating both STM and RTM on the same commodity processor. The experimental setup consisted of 4th generation of 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 (instruction and data), 256 KB L2 cache, and 8 MB of L3 cache. The operating system used is 64-bit Ubuntu Linux with 3.4.5-40 kernel. In order to access Intel's TSX intrinsic, -mrtm flag was used. All benchmarks are compiled using gcc 4.8.1. Sections 4.1.1 to 4.1.3 describe the general characteristics of each benchmark suite that was used for evaluation.

4.1.1 Stanford Transactional Applications for Multi-Processing (STAMP)

STAMP [2] is a well known and widely used benchmark suite for parallel computing. The input variables for each benchmark in STAMP can be configured. For all evaluations conducted on these benchmarks, the input variables consist of the maximum allowed parameters (non-simulated input parameters).

KMEANS: This benchmark represents a K-means algorithm that groups objects into 'K' number of clusters. The basis of this algorithm is to partition data into subsets. In this benchmark, there are three transactions. The transaction sizes in KMEANS are relatively small and so are the read and write sets.

GENOME: This benchmark represents reconstructing and matching DNA segments. The structure of this benchmark consists of five transactions that range from medium to large transactional sizes. The working-sets (read/write) in this benchmark have moderate size.

LABYRINTH: This benchmark represents the structure of a three-dimensional maze. Each thread essentially attains a start and an end point of the maze, and connects a path through all grid points. The structure of this benchmark consists of three transactions in which the execution time of one transaction dominates the other two. The transaction sizes range from medium to large and the working set size is large.

SSCA2 (Scalable Synthetic Compact Applications 2): This benchmark represents the construction of an array data structure for security based applications. The structure of this benchmark consists of only one small sized transaction as well as a small working set size.

YADA (Yet Another Delaunay Application): This benchmark represents Ruppert's algorithm [24] for mesh refinement data structure. The structure of this benchmark consists of five transactions ranging from small to large sizes. The working set size also ranges from medium to large sizes.

4.1.2 NAS Parallel Benchmarks

This benchmark suite was introduced in 1994 by Ames Research Center of NASA and was developed for performance evaluation of highly parallel supercomputers [4]. These benchmarks mimic the computation and data structures of CFD (computational fluid dynamics) applications [19]. This benchmark suite was used in this thesis to further enhance the spectrum of transactional memory applications.

Conjugate-Gradient: This benchmark represents gird computations for unstructured eigenvalues. The structure of this benchmark consists of six transactions ranging from small to large sizes. The working set size is fairly small throughout the six transactions.

Multi-Grid: This benchmark represents the testing of short and long distance data communications. The structure of this benchmark consists of two transactions in which one of the transactions is very small and the other is medium sized. The working set size is fairly small for both transactions.

Embarrassingly Parallel: This benchmark represents calculation of floating-point data structures without significant inter-processor communication. The structure of this benchmark consists of three transactions. The transaction sizes range from medium to large. The working set size remains fairly moderate in size.

4.1.3 DiscoPoP Benchmark Suite

This benchmark suite was developed for a tool to automatically find potential parallelism in sequential programs [37]. This tool is called DiscoPoP which is able to find parallelism between code regions with subjective granularity. The set of benchmarks introduced in DiscoPoP is also used for the evaluation of the proposed adaptive system. The benchmarks used are the following:

- Mandelbrot
- Light_ Propagation
- Monte Carlo
- Artificial Neural Network Training

Each of these benchmarks only consists of one transaction. The transaction size however ranges from small to medium. The working set size is considerably small when compared to both NAS and STAMP benchmarks. These sets of benchmarks were used to further enhance the prediction of the decision tree. The adaptive system must be able to work with a wide variety of transactional applications, including applications that have minimal transaction sizes and minimal working set sizes.

4.2 RTM vs. STM Performance Evaluation

The first set of experiments are based on evaluation of RTM and TinySTM on 12 benchmarks taken from Stamp [2], NAS [4], and DiscoPop [37] benchmark suites. This evaluation is primarily conducted to compare the performances of the two systems. Figure 4.1 represents a normalized comparison graph between RTM and TinySTM. In each benchmark, the number for threads varies between two and eight. In Figure 4.1, measurement reading greater than one favors TinySTM while less than one favors RTM. There is a vast discrepancy between both systems, primarily due to the transaction characters within a given benchmark such as transaction size, write-set size and read-set size.

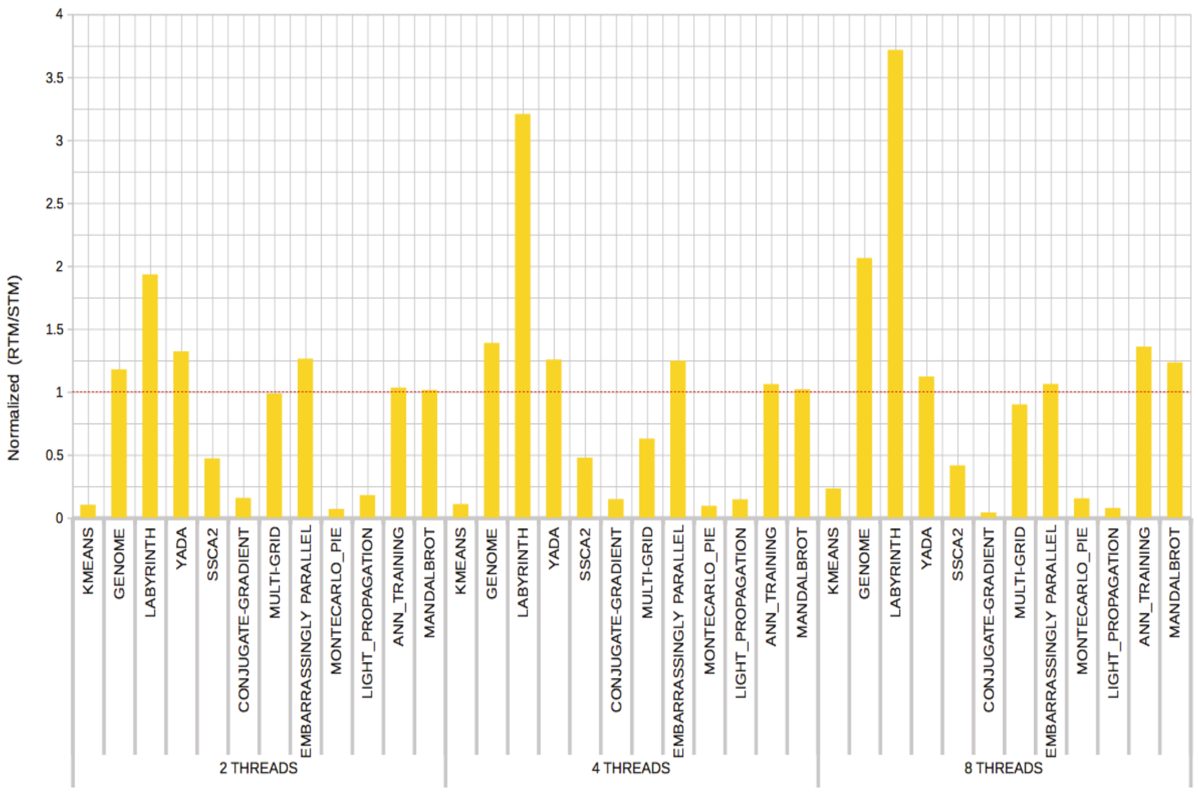


Figure 4.1: Normalized Transactional Execution time of RTM relative to TinySTM.

In small benchmarks where working set of the benchmark fits in the L1 cache, i.e. Montecarlo, Light_Propagation, KMEANS, SSCA2, Conjugate-Gradient, RTM outperforms TinySTM. In contrast, TinySTM outperforms RTM in benchmarks consisting of larger transaction sizes, i.e. Labyrinth, Genome, YADA, Ann_Training, Mandalbrot. The number of transactions within a benchmark varies and the characteristics from one transaction to another also vary. By introducing our adaptive system, we will be able to switch between RTM and TinySTM within a benchmark and achieve better performance.

4.3 RTM vs. STM Energy Expenditure Evaluation

An important aspect of computational performance is energy efficiency. With modern technology (laptops, cellphones, tablets, etc.) relying heavily on battery power, it is essential to expend an efficient amount of energy as possible. Energy expenditure was accurately measured using Intel's runtime average power limit monitor (RAPL) [16], calculated in milli-joules (mJ). RAPL relies on a set of hardware counters inside the processor, which provides energy and power consumption information.

The energy measurements are first taken for each TM system and an analysis is made. One of the major advantages of RTM over TinySTM is energy efficiency. RTM is more energy efficient than TinySTM as RTM exploits hardware resources and does not incur the software overhead of TinySTM. This experiment was conducted to see the extent of how energy efficient RTM is over TinySTM. The same 12 benchmarks from STAMP, NAS and DiscoPoP are used for this evaluation.

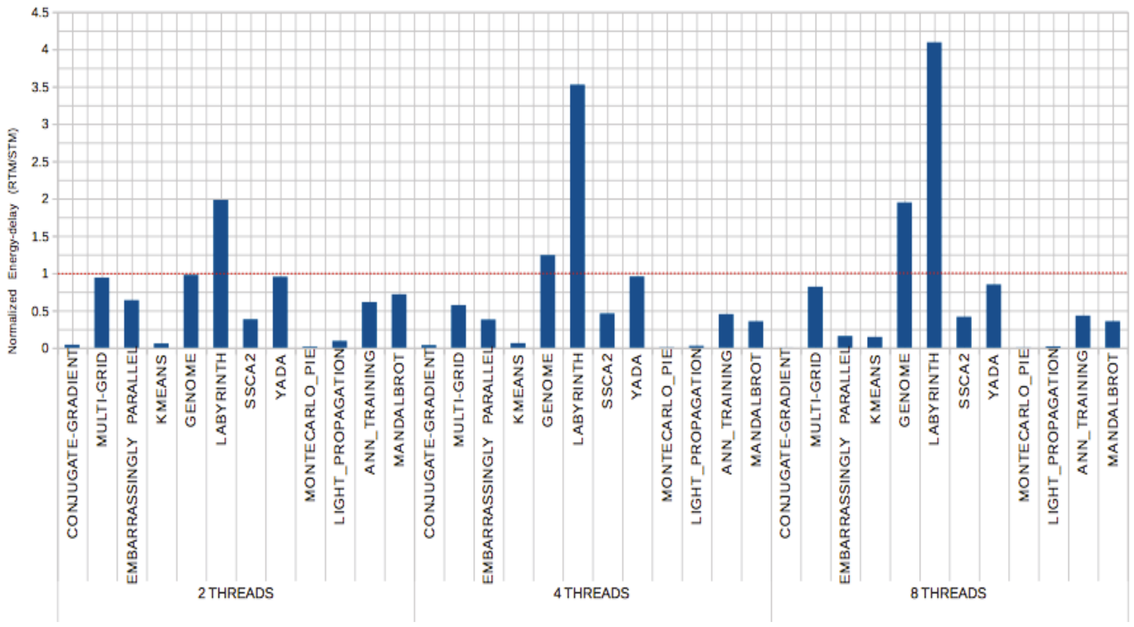


Figure 4.2: Normalized Energy-delay of RTM relative to TinySTM.

Figure 4-2 represents energy-delay comparison between RTM and TinySTM. The energy delay measurement is calculated by the energy consumption multiplied by the transactional execution time. To take into account the impact of both energy and performance, we use energy-delay to compare adaptive system with RTM and TinySTM. RTM is much more energy efficient than STM for all benchmarks except for benchmarks GENOME and LABYRINTH. This is primarily due to the benchmarks characteristics as well as the structure of RTM. Although, RTM is generally much more energy efficient compared to STM, the structure of RTM can lead to excess wasted work. When RTM aborts, the retry sequence is initiated where it will keep retrying the aborted transaction.

Once the retry threshold is reached, the transaction will be executed using the fallback policy (global lock). This results in wasted work as the abort prone transaction is retried unsuccessfully. Another important limitation of RTM is capacity induced aborts. No matter how many times the transaction is retried, the hardware limitations restrict it from successfully committing. 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 incorporates STM meaning that the energy efficiency readings compared to RTM does not result in efficiency. Table 4.1 and 4.2 depict the characteristics of benchmark Genome and Labyrinth, respectively. (Further analysis of all benchmarks is in Appendix-A).

Table 4.1: Characteristics of Benchmark LABYRINTH at two threads

TX #	STM Time(ms)	RTM Time(ms)	Read-set Size	Write-set Size	Tx Size	Abort Ratio(STM)	Abort Ratio(RTM)	Write Ratio	Capacity Abort-Ratio
TX1	0	0	4108	512	134	0	0	0.1109	0
TX2	77851	150512	1151846	1810368	254	0.04119	0.7470355731	0.6112	0.9596560
TX3	0	0	12	8	61	0	0	0.3636	0

Table 4.2: Characteristics of Benchmark GENOME at two threads.

TX #	STM Time(ms)	RTM Time(ms)	Read-set Size	Write-set Size	Tx Size	Abort Ratio(STM)	Abort Ratio(RTM)	Write Ratio	Capacity Abort-Ratio
TX1	6064	7359	41992177	32652	259	2.14575E-006	0.2037975743	0.0077	0.5876806
TX2	2	1	21728	16321	116	0	0.0001837785	0.4289	0.3666666
TX3	2493	2774	40543510	2057244	536	0.0097530404	0.2562808218	0.0482	0.3399366
TX4	4	3	52050	32642	133	0	0.1450497643	0.3854	0.0003611
TX5	8	2	107612	81600	154	0	0.0037543224	0.4312	0.1315789

From these tables, the results show that all transactions are different from one another in terms of transaction size and working set size. In benchmark GENOME, the average capacity abort ratio (only for TX1 and TX2, due to majority of transactional load) is approximately 46.8% out of the total number of aborts. The capacity abort results in slowdown for RTM when compared to STM. For benchmark Labyrinth, only one of the three transactions has the majority of the transactional load. The capacity abort ratio for that transaction is 95.9% of the total number of aborts. This severely hampers RTM's performance, as it wastes a lot of work by retrying unnecessarily and executing the fallback path. On the other side, for STM, the total abort ratio is very small at 4.11%. Figure 4.3 depicts the distribution of transactional aborts for benchmark GENOME. Generally, as the thread count increases from 2 to 8, the capacity aborts increase from 46.4% to 74.6%.

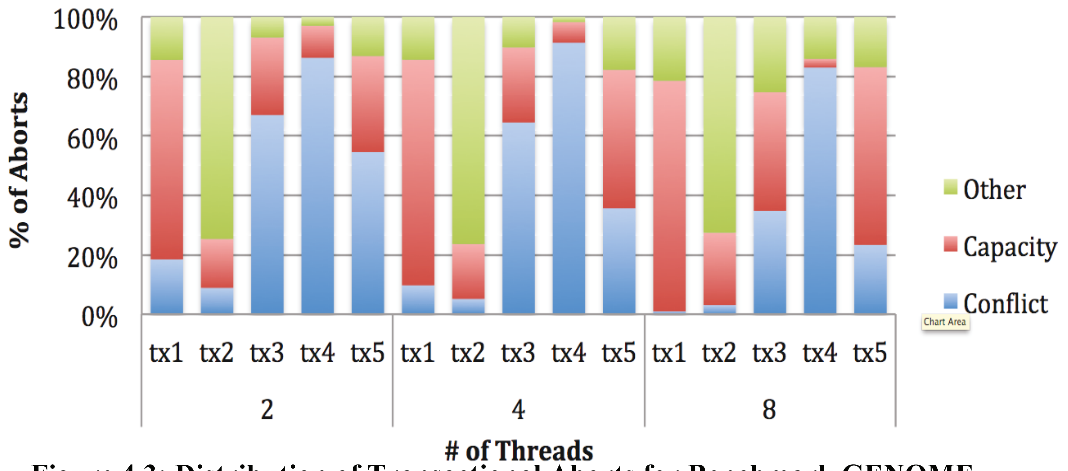


Figure 4.3: Distribution of Transactional Aborts for Benchmark GENOME

4.3 Evaluation of Adaptive system

The experimental analysis of the adaptive system is to compare the results with baseline TinySTM and baseline RTM. This evaluation consists of both transactional execution time as well as energy delay measurements. For evaluation, the benchmarks from the testing phase are used. This includes benchmarks Conjugate-Gradient, Multi-Grid, KMEANS, SSCA2, Ann_Training and Mandalbrot. The primary objective of the testing benchmarks is to strictly use the decision tree predictions. Therefore, the focus was on attaining a prediction based on the training benchmarks then applying the prediction to another set of benchmarks (testing benchmarks).

4.3.1 Adaptive system vs. TinySTM

This section provides experimental analysis between the proposed adaptive system and TinySTM. Figure 4.4 depicts Normalized transactional execution time (speedup) 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 Mandalbrot have a normalized speedup value of 1 which indicates that the prediction used for the adaptive system heavily favored TinySTM. Overall, as the thread count increases, there is little difference in speedup. On average, speed-up is 34.31%, 34.44%, and 34.35% for 2, 4 and 8 threads, respectively. It is important to note that the decision tree prediction is not always correct, as a few predictions are inaccurate. Yet, the performance gains of the proposed adaptive system are very promising when compared to TinySTM. A thorough analysis of the decision tree prediction for each testing benchmark is found in Appendix B.

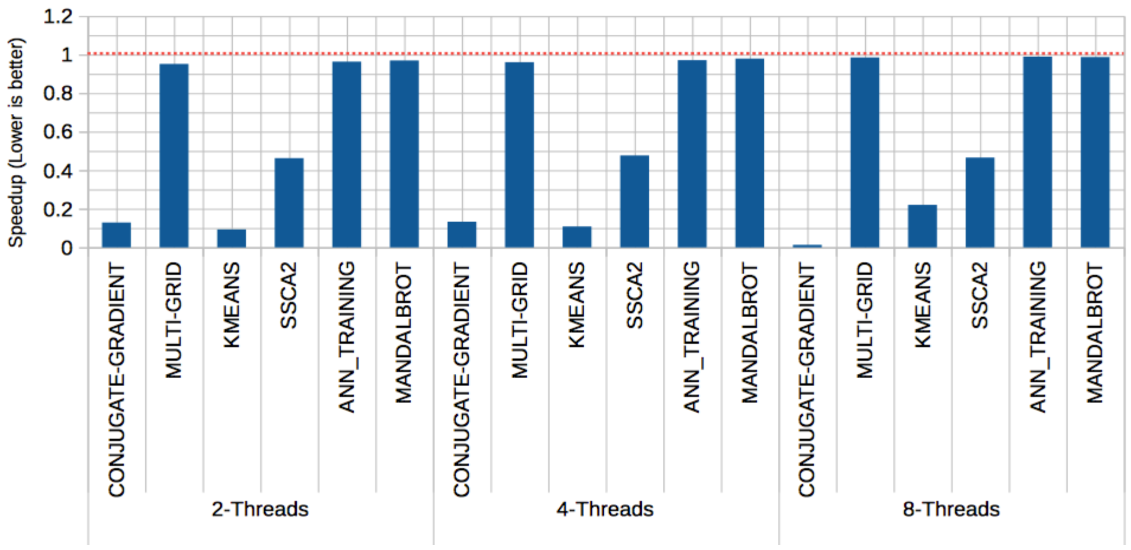


Figure 4.4: Normalized Speedup comparison between adaptive system and TinySTM.

The next evaluation is based on the energy delay measurements. Figure 4.5 depicts Normalized energy-delay comparison between the adaptive system and TinySTM. Once again for this evaluation, only the benchmarks in the testing benchmarks 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 and a value greater than one depicts energy deficiency. In all the testing benchmarks, our adaptive system is 42.11% more energy efficient than TinySTM. This is a significant difference of energy consumption when compared to baseline TinySTM. The reason for this substantial energy efficiency is that for certain benchmarks that consist of low/medium sized transactions and working set sizes, by implementing these transaction in RTM, the adaptive system is able to save energy. The benchmarks that show significant energy efficiency are (portrays overall energy efficiency percentage):

- Conjugate-Gradient → 94.78%
- KMEANS → 91.76%
- SSCA2 → 55.81%

The rest of the benchmarks relatively have low to moderate energy efficiency. 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. (Further analysis of energy expenditure for all benchmarks is found in Appendix D.)

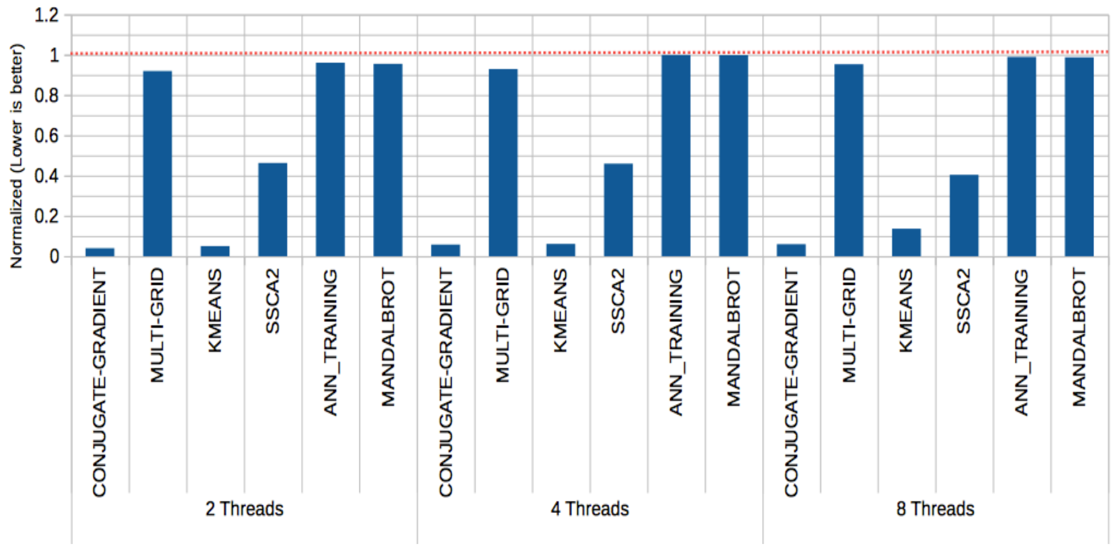


Figure 4.5: Normalized Energy-delay comparison between adaptive system and TinySTM.

4.3.2 Adaptive system vs. RTM

This section provides experimental analysis between the proposed adaptive system and RTM. Figure 4.6 depicts Normalized transactional execution time (speedup) between the adaptive system and RTM. The benchmarks that have a normalized speedup less than one indicate that the adaptive system achieves speedup.

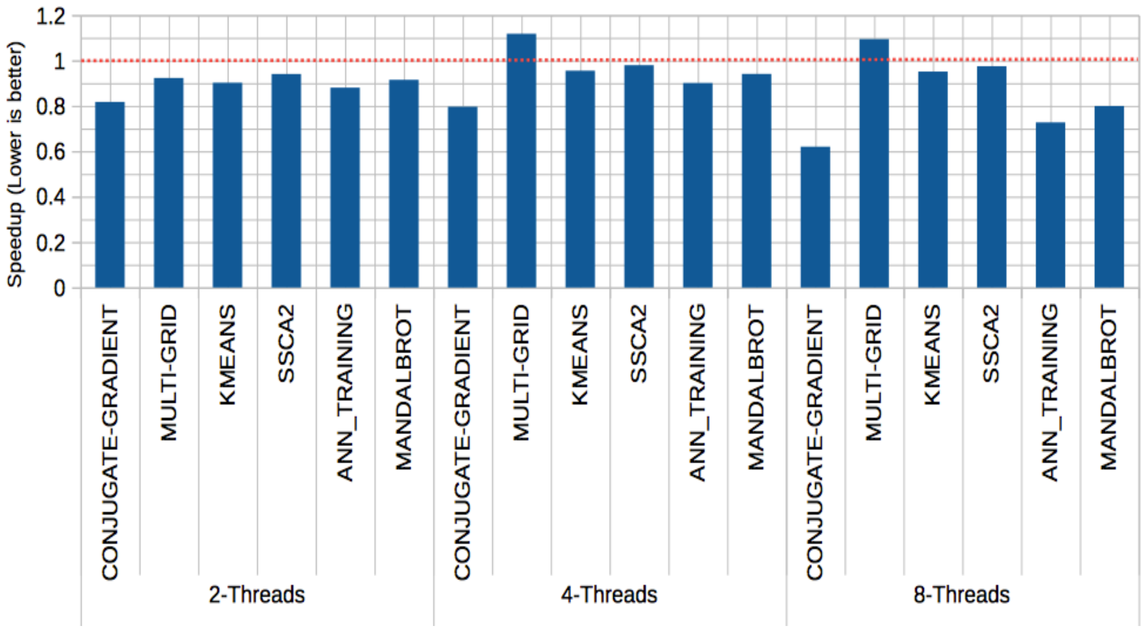


Figure 4.6: Normalized Speedup comparison between adaptive system and RTM.

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 4.3 shows transaction parameters of Multi-Grid. At 4 threads, Multi-Grid has a better execution time for RTM, but due to the decision tree's prediction, the STM system was used. 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.

Table 4.3: Transaction parameters and execution time for Multi-Grid benchmark when the number of threads is four.

TX#	STM Time(ms)	RTM Time(ms)	Read-set Size	Write-set Size	TX Size	Write Ratio	Decision tree Prediction	Optimum System
TX1	120	60	64	64	130	0.5	RTM	RTM
TX2	18818	16990	8008	8008	276	0.5	STM	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 (please refer to appendix A.2). 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. On the contrary, at 8 threads, benchmarks Conjugate-Gradient, Ann_Training and Mandalbrot demonstrate good speedup when compared to the baseline RTM. On average, the proposed adaptive system has speedup of 5.88%, 5.16% and 11.79% for 2, 4 and 8 threads, respectively.

The next evaluation is based on the energy-delay measurements. Figure 4.7 depicts normalized energy-delay comparison between the adaptive system and RTM. The proposed adaptive system is not energy efficient when compared to RTM. This is primarily 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.

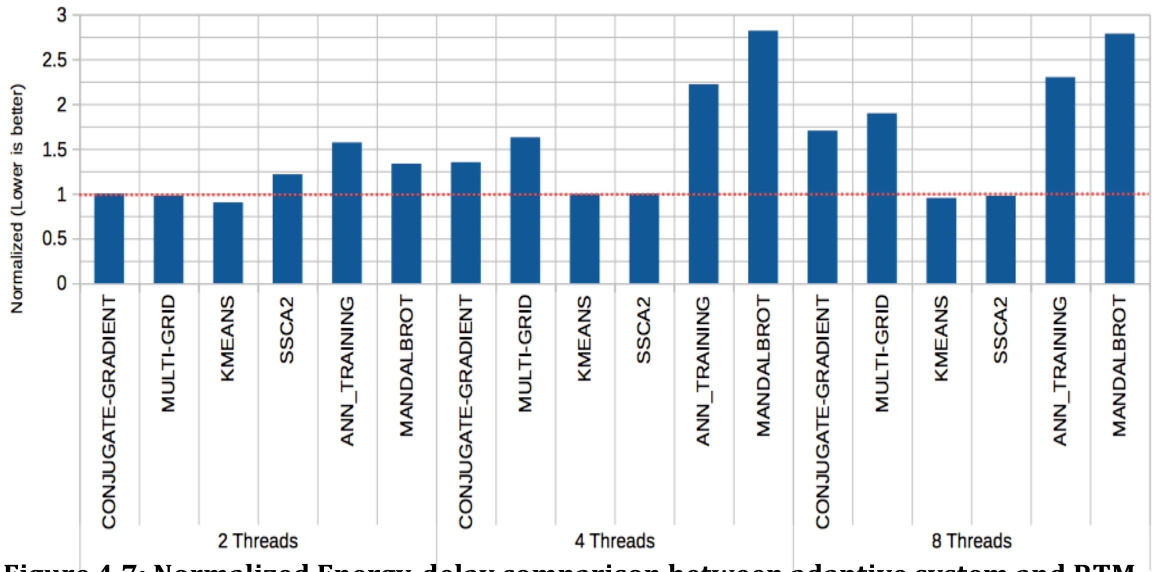


Figure 4.7: Normalized Energy-delay comparison between adaptive system and RTM

4.3.3 Decision Tree Predictions for Testing Benchmarks

This section reviews the decision tree prediction that was used for each of the testing benchmarks. During the training phase, the system that executed the fastest was included as the input parameter for the decision tree. For the testing phase, the decision tree does not predict correctly all the time.

These tables show (4.4, 4.5, 4.6, 4.7, and 4.8) that the proposed adaptive system is able to achieve speedup in all benchmarks (except Multi-Grid, explanation is in section 4.3) when compared to RTM or TinySTM. These tests also show exactly which transaction yields the majority of the application's workload. For example, in table 4.4 (benchmark Conjugate-Gradient) TX2 takes the majority of the transactional execution time when compared to the other transactions. If the decision tree predicts incorrectly, this can lead to performance slowdown for the adaptive system. This shows that the accuracy of the decision tree is crucial to achieve speedup for applications. In SSCA2 (table 4.7), there are 3 transactions in total but only one out of the tree transactions has the application's entire workload. (Further analysis of energy expenditure for all benchmarks is found in Appendix D.)

Table 4.4: Benchmark Conjugate-Gradient comparing Decision Tree prediction with Optimum system

TX#	STM Time(ms)	RTM Time(ms)	Adap. Time(ms)	Speedup (Baseline_STM)	Speedup (Baseline_RTM)	D.T prediction	Optimum prediction
TX1	4	21	19	4.75	0.9047619048	RTM	STM
TX2	83391	9664	9473	0.1135973906	0.9802359272	RTM	RTM
TX3	97	809	489	5.0412371134	0.6044499382	STM	STM
TX4	14	2	28	2	14	STM	RTM
TX5	4	20	19	4.75	0.95	RTM	STM
TX6	172	1873	170	0.988372093	0.090763481	STM	STM

Table 4.5: Benchmark Multi-Grid comparing Decision Tree prediction with Optimum system

TX#	STM Time(ms)	RTM Time(ms)	Adap. Time(ms)	Speedup (Baseline_STM)	Speedup (Baseline_RTM)	D.T prediction	Optimum prediction
TX1	39	35	36	0.9230769231	1.0285714286	RTM	RTM
TX2	20738	13026	19919	0.9605072813	1.5291724244	STM	RTM

Table 4.6: Benchmark KMEANS comparing Decision Tree prediction with Optimum system

TX#	STM Time(ms)	RTM Time(ms)	Adap. Time(ms)	Speedup (Baseline_STM)	Speedup (Baseline_RTM)	D.T prediction	Optimum prediction
TX1	6422	640	642	0.0999688571	1.003125	RTM	RTM
TX2	102	0	0	0	0	RTM	RTM

Table 4.7: Benchmark SSCA2 comparing Decision Tree prediction with Optimum system

TX#	STM Time(ms)	RTM Time(ms)	Adap. Time(ms)	Speedup (Baseline_STM)	Speedup (Baseline_RTM)	D.T prediction	Optimum prediction
TX1	0	0	0	0	0	n/a	n/a
TX2	0	0	0	0	0	n/a	n/a
TX3	5584	2663	2662	0.4767191977	0.9996244837	RTM	RTM

Table 4.8: Benchmark ANN_TRAINING comparing Decision Tree prediction with Optimum system

TX#	STM Time(ms)	RTM Time(ms)	Adap. Time(ms)	Speedup (Baseline_STM)	Speedup (Baseline_RTM)	D.T prediction	Optimum prediction
TX1	42698	45335	42656	0.9990163474	0.9409065843	STM	STM

Table 4.9: Benchmark MANDALBROT comparing Decision Tree prediction with Optimum system

TX#	STM Time(ms)	RTM Time(ms)	Adap. Time(ms)	Speedup (Baseline_STM)	Speedup (Baseline_RTM)	D.T prediction	Optimum prediction
TX1	18825	19207	18785	0.997875166	0.9780288437	STM	STM

Chapter 5

Conclusion

CMPs have become the main architecture of general-purpose computing. This made development of efficient parallel programs a necessity in order to increase performance. Transactional memory (TM) has been established as a simple and effective parallel programming paradigm. TM has become progressively widespread especially with Hardware transactional memory implementations becoming increasingly available. This thesis proposes an adaptive system that exploits both STM and HTM at transaction granularity. This chapter concludes the thesis and offers the potential future work that can enhance performance of TM programs further.

5.1 Summary of Contributions

In a typical parallel application, the characteristics of a transaction vary immensely. This leads to the discovery that there is no single TM system that works well across all parallel applications. The primary goal of this thesis is to improve the performance of parallel applications by combining the benefits of both RTM and TinySTM. With the proposition of the adaptive system, it is possible to switch between RTM and TinySTM at transaction granularity. A synchronization technique is developed in order to seamlessly switch between RTM and TinySTM based on the characteristics of a transaction. By exploiting the decision tree prediction module, it is possible 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 (STAMP, NAS and DiscoPoP) 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 (2, 4 and 8). These predictions are then evaluated on the testing phase which reveal that the adaptive system is able to improve transactional execution time and energy-delay.

5.2 Future Work

With the development of the adaptive system, there are issues that can be improved with further optimizations.

1) For this thesis, the training dataset of the decision tree was limited to 6 benchmarks (the other benchmarks are used for testing) that ranged from small to large transaction sizes and working set sizes. By incorporating additional benchmark suites for the training phase, it is possible to improve the accuracy of the decision tree prediction module.

2) The other opportunity for future work is combining adaptive system with the technique proposed by Yang et al. [36] (further information can be found in section 2.4). By implementing the optimization techniques introduced in [36] in conjunction with the adaptive system, it is possible to optimize STM and RTM separately based on the transactional characteristics (such as transaction size, read-set size, write-set size, etc.). This will further enhance the accuracy of the predictions made by the decision tree as well as increase the performance of the application.

Appendix

Benchmark Abbreviations used are the following:

NAS benchmark suite

CG – Conjugate Gradient

MG – Multi-Grid

EP – Embarrassingly parallel

DiscoPoP benchmark suite

09 – MONTECARLO_PIE

10 - LIGHT_PROPAGATION

11 - ANN_TRAINING

12 - MANDALBROT

A. Total Analysis of Benchmarks for threads 2, 4 and 8

2 THREADS

BENCHMARK	PC_TXN	STM Time(ms)	RTM Time(ms)	Read-set_Size	Write-set_Size	Transaction_Size	Abort_Ratio (STM)	Abort_Ratio (RTM)	Write_Ratio	Capacity_Abort_Ratio (RTM)
KMEANS	402748	5206	586	14417920	49020928	300	0.1643389008	0.0926543156	0.7727272727	0.0479120656
	4028b2	88	0	961180	961180	117	0.0007641008	0.0059909013	0.5	0.6442351511
	402956	0	0	88	88	103	0	0	0.5	0
GENOME	402986	6064	7359.77	41992177	32652	259	2.14575851490623E-006	0.2037975743	0.0007769693	0.5876806917
	402c60	2	1	21728	16321	116	0	0.0001837785	0.4289468843	0.6666666667
	_402e05	2493	2774	40543510	2057244	536	0.0097530404	0.2562808218	0.0482912579	0.3399366108
	402f3e	4	3	52050	32642	133	0	0.1450497643	0.3854201105	0.0003611412
	40321d	8	2	107612	81600	154	0	0.0037543224	0.4312622878	0.1315789474
LABYRINTH	403d7e	0	0	4108	512	134	0	0	0.1109666233	0
	403ec8	77851	150512	115184	181036	214	0.0411985019	0.7470355731	0.6112005399	0.9596560847
	_403e59	0	0	16	8	61	0	0	0.3636363636	0
SSCA2	402acd	0	0	2	2	158	0	0	0.5	0
	403173	0	0	2	2	118	0	0	0.5	0
	403b0d	5632	2658	22362279	44724558	452	7.69146458945846E-006	0.0032987627	0.6666666667	0.011984543
VACATION	40187a	54790	22877	1504567408	22584082	699	0.0016807612	0.4962845821	0.0147883705	0.6709735203
	_401e16	3518	1904	76090891	5746557	537	0.0036350366	0.6698811148	0.0702191618	0.5071978028
	401ba2	1223	574	25323336	1393626	155	0.0006824929	0.2603699334	0.0521625924	0.6742283427
YADA	_4053e6	243	102	2511729	1214327	101	0.1557759763	0.1038268133	0.3259014357	0.9978060029
	4054c7	362	49	578732	0	115	0.9093182791	0	0	0
	405571	18347	25036	102677075	24047000	626	0.9701766885	0.7493013556	0.1897587337	0.0146247631
	405619	45	22	0	445755	95	6.73010916237061E-006	0.000019401	1	1
	4056c3	125	61	1120430	502102	109	0.0222441083	0.011626916	0.3094558382	0.9948689756
CG	402add	3	15	0	0	146	0	0.0014835102	0	0.9038461538
	402d53	42901	5277	2267582	2978034	580	0.3623419973	0.6799798386	0.5677186435	0.0437713224
	402fa1	95	405	0	1133791	254	0	0.2018376244	1	0.000011229
	403259	6	2	28000	28000	206	0.7782283614	0.1591591592	0.5	0
	4033d5	3	11	2	2	138	0	0.75	0.5	1
	4037c8	153	1073	0	0	157	0	0.75	0	1
MG	403f65	21	20	16	16	130	0.3333333333	0.75	0.5	0
	404554	10689	10093	2002	2002	276	0	0.076182708	0.5	0.0496898691
EP	401d4f	0	0	4	4	228	0	0	0.5	0
	4020d4	82106	72497	8192	8192	463	0.9999805467	0.7456137627	0.5	0.0062887843
	4022a1	26672	64953	16384	16384	701	0	0.7455189339	0.5	0.4155173132
DPOP_09	40164b	33889	2372	75000	75000	241	0.362463502	0.1465912416	0.5	0.0432237301
DPOP_11	401b11	28635	5158	38755450	0	210	0.2494688428	0.18280796	0	0.0691665432
DPOP_12	401754	25610	26486	4	12006	821	0.0197628094	0.42089879	0.9996669442	0.6284843072
DPOP_13	401bae	8424	8553	2	2	598	0.0927106095	0.233375253	0.5	0.506916652

4 THREADS										
BENCHMARK	PC_TXN	STM Time(ms)	RTM Time(ms)	Read-set_Size	Write-set_Size	Transaction_Size	Abort_Ratio (STM)	Abort_Ratio (RTM)	Write_Ratio	Capacity_Abort_Ratio (RTM)
KMEANS	402748	6422	640	16056320	54591488	300	0.713292697	0.253015859	0.7727272727	0.0586671172
	4028b2	102	0	1070405	1070405	117	0.1138642672	0.0283316171	0.5	0.4943088154
	402956	0	0	196	196	103	0.0967741935	0	0.5	0
GENOME	402986	6352	8875	41992319	32694	259	7.15198614231165E-005	0.2876113725	0.0007779653	0.9333249476
	402c60	2	1	26921	16321	116	0	0.0005511329	0.3774339762	0.4444444444
	_402e05	2194	2991	37219971	2058326	536	0.0129078543	0.2278967679	0.0524036467	0.4387203701
	402f3e	4	3	52050	32642	133	0	0.1226683868	0.3854201105	0.0096406661
	40321d	8	1	107591	81615	154	0	0.0090087139	0.4313552424	0.7672727273
LABYRINTH	403d7e	0	0	4108	512	134	0	0	0.1108225108	0
	403ec8	93700	300580	115184	181036	214	0.0791366906	0.744638404	0.6111538721	0.9564634963
	_403e59	0	0	16	8	61	0	0	0.3333333333	0
SSCA2	402acd	0	0	4	4	158	0	0	0.5	0
	403173	0	0	4	4	118	0	0	0.5	0
	403b0d	5584	2663	22362277	44724554	452	3.58626824150646E-005	0.0021343047	0.6666666667	0.0943131926
VACATION	40187a	61372	30193	1504966208	22586779	699	0.0061596699	0.5794654643	0.0147862491	0.766526462
	_401e16	3843	2510	76105089	5749087	537	0.0072882028	0.5051212887	0.0702357202	0.1041553859
	401ba2	1365	769	25294852	1391929	155	0.0063343471	0.4086016639	0.0521579954	0.7617587195
YADA	_4053e6	291	113	2525298	1219387	101	0.353491829	0.0097723213	0.3256313949	0.9665616246
	4054c7	523	48	580197	0	115	0.9332216595	1.72770653867817E-005	0	0.8
	405571	39833	51061	103961525	24145158	626	0.9828129855	0.7490248419	0.1884769587	0.1107261368
	405619	52	24	0	464996	95	9.24653631200824E-005	0.0001498392	1	0.9166666667
	4056c3	144	66	1127133	505601	109	0.072271513	0.0184724762	0.3096652608	0.9746737447
CG	402add	4	21	0	0	146	0	0.0172404111	0	0.1530944625
	402d53	83391	9664	2267276	3056056	580	0.391903403	0.6905205223	0.5740870568	0.0434589992
	402fa1	97	809	0	1133638	254	0	0.7935343749	1	3.4266738522979E-005
	403259	14	2	28000	28000	206	0.9121816585	0.2230509679	0.5	0.5
	4033d5	4	20	4	4	138	0	0.75	0.5	1
	4037c8	172	1873	0	0	157	0	0.7142857143	0	0.6
MG	403f65	39	35	32	32	130	0.6	0.6923076923	0.5	0
	404554	20738	13026	4004	4004	276	0.3188159238	0.1146387092	0.5	0.0954764852
EP	401d4f	0	0	8	8	228	0	0	0.5	0
	4020d4	192968	142480	8192	8192	463	0.9999955317	0.7463227325	0.5	0.0095016804
	4022a1	28349	133555	16384	16384	701	0	0.746628727	0.5	0.4029826015
DPOP_09	40164b	78405	7369	125000	125000	241	0.5296801836	0.29038701	0.5	0.1248248838
DPOP_11	401b11	92836	13568	64588342	0	210	0.490647486	0.26316142	0	0.1652646203
DPOP_12	401754	42698	45335	8	24012	821	0.0497419354	0.41913906	0.9996669442	0.7483850589
DPOP_13	401bae	18825	19207	4	4	598	0.1028415961	0.4767856833	0.5	0.6387203701

8 THREADS										
BENCHMARK	PC_TXN	STM Time(ms)	RTM Time(ms)	Read-set_Size	Write-set_Size	Transaction_Size	Abort_Ratio (STM)	Abort_Ratio (RTM)	Write_Ratio	Capacity_Abort_Ratio (RTM)
	402748	12881	2673	15400960	52363264	300	0.9265017641	0.5770788085	0.7727272727	0.0491449124
KMEANS	4028b2	192	0	1026715	1026715	117	0.7051197325	0.0721947775	0.5	0.2118493461
	402956	0	0	376	376	103	0.4311649017	0	0.5	0
	402986	8815	24503	41993415	32864	259	9.5832776096731E-005	0.4805312158	0.0007819869	0.9753121262
	402c60	2	1	31114	16321	116	0	0.0016515782	0.3440708338	0.8148148148
GENOME	_402e05	4315	3407	40263763	2063398	536	0.6550935514	0.1812389913	0.0487487928	0.5158542211
	402f3e	4	2	51915	32642	133	0.0165109973	0.0935799178	0.3860354554	0.0492581602
	40321d	8	1	107570	81610	154	0.0090736373	0.0091710065	0.431388096	0.925
	403d7e	0	0	4120	512	134	0	0	0.1105354059	0
LABYRINTH	403ec8	160384	596181	115260	181378	214	0.1674796748	0.743871936	0.6114456004	0.9952925353
	_403e59	0	0	34	16	61	0	0	0.32	0
	402acd	0	0	8	8	158	0	0	0.5	0
SSCA2	403173	0	0	8	8	118	0	0	0.5	0
	403b0d	7180	2986	22362279	44724558	452	0.0193154125	0.0030602841	0.6666666667	0.2000145677
	40187a	85142	74549	1506135671	22599539	699	0.0524959346	0.5992925883	0.0147831612	0.8141055188
VACATION	_401e16	5352	5993	76715350	5797560	537	0.0182928247	0.6555147251	0.0702624595	0.6839173208
	401ba2	1842	1898	25321248	5797560	155	0.0067240669	0.4517710844	0.1863040512	0.7729247272
	_4053e6	529	115	2538569	1224457	101	0.6926425984	0.0318358791	0.3253915865	0.9698592733
	4054c7	834	48	581558	0	115	0.9353990254	2.41731488116307E-005	0	0.8571428571
YADA	405571	91087	103812	104568795	24242674	626	0.9864962343	0.8308737412	0.1882027601	0.5000029081
	405619	80	24	0	481786	95	0.0019307339	0.0001612107	1	0.7721518987
	4056c3	263	70	1131533	507161	109	0.4363099653	0.0221743695	0.3094909727	0.9867698677
	402add	7	28	0	0	146	0	0.0190308024	0	0.8762886598
	402d53	1207360	19417	2266606	3281599	580	0.1492472889	0.7156772633	0.5914703945	0.042444403
CG	402fa1	114	2230	0	1133303	254	0	0.3584304513	1	8.96346025939475E-005
	403259	20	2	28000	28000	206	0.9023089966	0.1465912416	0.5	0.0099800399
	4033d5	12	39	8	8	138	0	0.75	0.5	0.9166666667
	4037c8	271	3293	0	0	157	0	0.7419354839	0	0.3913043478
MG	403f65	120	60	64	64	130	0.7777777778	0.724137931	0.5	0.0952380952
	404554	18818	16990	8008	8008	276	0.408392435	0.310687302	0.5	0.1945600361
	401d4f	0	0	16	16	228	0	0	0.5	0
EP	4020d4	580309	282834	8192	8192	463	0.9999814146	0.7466522344	0.5	0.0069171188
	4022a1	33966	270128	16384	16384	701	0.0017060687	0.7466914038	0.5	0.4178399867
DPOP_09	40164b	1647025	25350	225000	225000	241	0.848361268	0.34587828	0.5	0.3926769231
DPOP_11	401b11	560846	43037	116244833	0	210	0.9018109976	0.6968257648	0	0.67320735
DPOP_12	401754	70874	96391	16	48024	821	0.0577078194	0.7726332502	0.9996669442	0.8148148148
DPOP_13	401bae	54520	67303	8	8	598	0.1201764086	0.6565147251	0.5	0.7148148969

B. Decision tree analysis for all Benchmarks

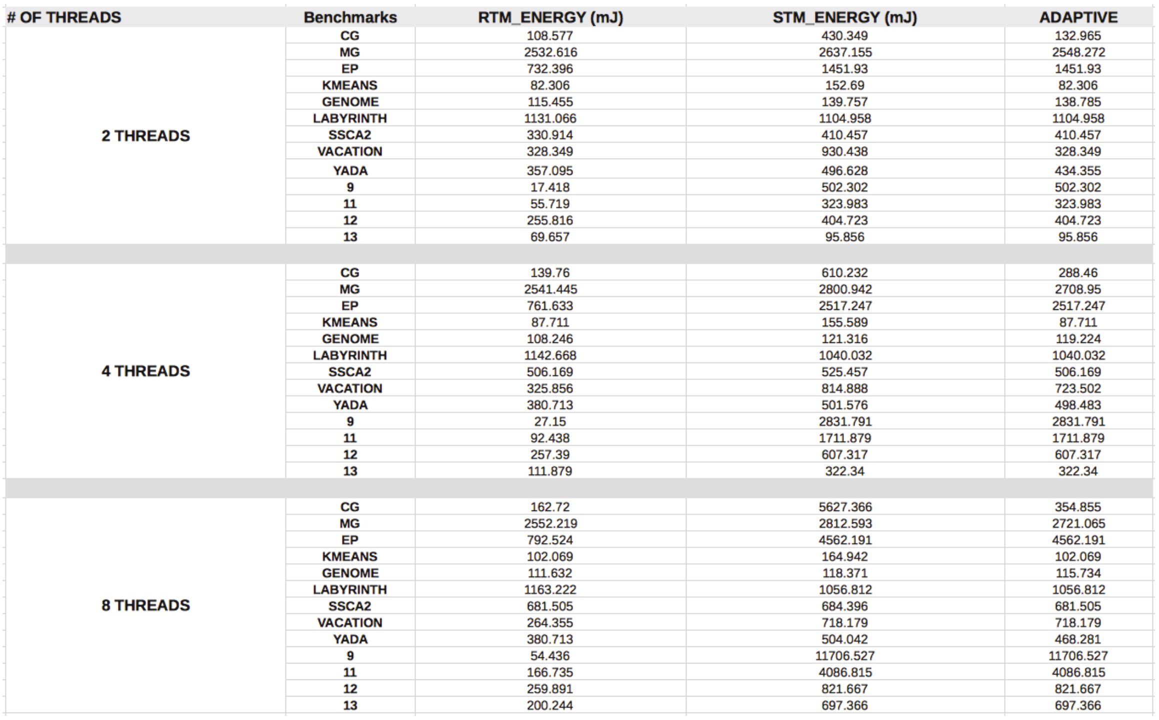
2 THREADS									
Benchmarks	PC_TXN	STM Time(ms)	RTM Time(ms)	Read-set_Size	Write-set_Size	Txn_Size	Write_Ratio	D.T Prediction	Optimum System
KMEANS	402748	5206	586	14417920	49020928	300	0.7727272727	RTM	RTM
	4028b2	88	0	961180	961180	117	0.5	STM	
	402956	0	0	88	88	103	0.5	//	
GENOME	402986	6064	7359.77	41992177	32652	259	0.0007769693	STM	STM
	402c60	2	1	21728	16321	116	0.4289468843	RTM	RTM
	402e05	2493	2774	40543510	2057244	536	0.0482912579	STM	STM
	402f3e	4	3	52050	32642	133	0.3854201105	RTM	RTM
	40321d	8	2	107612	81600	154	0.4312622878	RTM	RTM
LABYRINTH	403d7e	0	0	4108	512	134	0.1109666233	//	
	403ec8	77851	150512	115184	181036	214	0.6112005399	STM	STM
	403e59	0	0	16	8	61	0.3636363636	//	
SSCA2	402acd	0	0	2	2	158	0.5	//	
	403173	0	0	2	2	118	0.5	//	
	403b0d	5632	2658	22362279	44724558	452	0.6666666667	RTM	RTM
VACATION	40187a	54790	22877	1504567408	22584082	576	0.0147883705	RTM	RTM
	401e16	3518	1904	76090891	5746557	537	0.0702191618	RTM	RTM
	401ba2	1223	574	25323336	1393626	155	0.0521625924	RTM	RTM
YADA	4053e6	243	102	2511729	1214327	101	0.3259014357	RTM	RTM
	4054c7	362	49	578732	0	115	0	RTM	RTM
	405571	18347	25036	102677075	24047000	626	0.1897587337	STM	STM
	405619	45	22	0	445755	95	1	RTM	RTM
	4056c3	125	61	1120430	502102	109	0.3094558382	RTM	RTM
CG	402add	3	15	0	0	146	0	RTM	STM
	402d53	42901	5277	2267582	2978034	580	0.5677186435	RTM	RTM
	402fa1	95	405	0	1133791	254	1	STM	STM
	403259	6	2	28000	28000	206	0.5	STM	RTM
	4033d5	3	11	2	2	138	0.5	RTM	STM
	4037c8	153	1073	0	0	157	0	STM	STM
MG	403f65	21	20	16	16	130	0.5	RTM	RTM
	404554	10689	10093	2002	2002	276	0.5	STM	RTM
EP	401d4f	0	0	4	4	228	0.5	///	
	4020d4	42106	72497	8192	8192	463	0.5	STM	STM
	4022a1	26672	64953	16384	16384	701	0.5	STM	STM
DPOP_09	40164b	33889	2372	75000	75000	241	0.5	STM	RTM
DPOP_11	401b11	28635	5158	38755450	0	210	0	STM	RTM
DPOP_12	401754	25610	26486	4	12006	821	0.9996669442	STM	STM
DPOP_13	401bae	8424	8553	2	2	598	0.5	STM	STM

4 THREADS									
Benchmarks	PC_TXN	STM Time(ms)	RTM Time(ms)	Read-set_Size	Write-set_Size	Txn_Size	Write_Ratio	D.T Prediction	Optimum System
KMEANS	402748	6422	640	16056320	54591488	300	0.7727272727	RTM	RTM
	4028b2	102	0	1070405	1070405	117	0.5	//	//
	402956	0	0	196	196	103	0.5	//	//
GENOME	402986	6352	8875	41993415	32864	259	0.0007779653	STM	STM
	402c60	2	1	31114	16321	116	0.3774339762	RTM	RTM
	402e05	2194	2991	40263763	2063398	536	0.0524036467	STM	STM
	402f3e	4	3	51915	32642	133	0.3854201105	RTM	RTM
	40321d	8	1	107570	81610	154	0.4313552424	RTM	RTM
LABYRINTH	403d7e	0	0	4108	512	134	0.1108225108	//	//
	403ec8	93700	300580	115184	181036	214	0.6111538721	STM	STM
	403e59	0	0	16	8	61	0.3333333333	//	//
SSCA2	402acd	0	0	4	4	158	0.5		//
	403173	0	0	4	4	118	0.5		//
	403b0d	5584	2663	22362277	44724554	452	0.6666666667	RTM	RTM
VACATION	40187a	61372	30193	1504966208	22586779	576	0.0147862491	STM	RTM
	401e16	3843	2510	76105089	5749087	537	0.0702357202	STM	STM
	401ba2	1365	769	25294852	1391929	155	0.0521579954	RTM	RTM
YADA	4053e6	291	113	2525298	1219387	101	0.3256313949	RTM	RTM
	4054c7	523	48	580197	0	115	0	RTM	RTM
	405571	39833	51061	103961525	24145158	626	0.1884769587	STM	STM
	405619	52	24	0	464996	95	1	RTM	RTM
	4056c3	144	66	1127133	505601	109	0.3096652608	RTM	RTM
CG	402add	4	21	0	0	146	0	RTM	STM
	402d53	83391	9664	2267276	3056056	580	0.5677186435	RTM	RTM
	402fa1	97	809	0	1133638	254	1	STM	STM
	403259	14	2	28000	28000	206	0.5	STM	RTM
	4033d5	4	20	4	4	138	0.5	RTM	STM
	4037c8	172	1873	0	0	157	0	STM	STM
MG	403f65	39	35	32	32	130	0.5	RTM	RTM
	404554	20738	13026	4004	4004	276	0.5	STM	RTM
EP	401d4f	0	0	8	8	228	0.5	//	//
	4020d4	192968	142480	8192	8192	463	0.5	STM	RTM
	4022a1	28349	133555	16384	16384	701	0.5	STM	STM
DPOP_09	40164b	78405	7369	125000	125000	241	0.5	RTM	RTM
DPOP_11	401b11	92836	13568	64588342	0	210	0	RTM	RTM
DPOP_12	401754	42698	45335	8	24012	821	0.9996669442	STM	STM
DPOP_13	401bae	18825	19207	4	4	598	0.5	STM	STM

8 THREADS

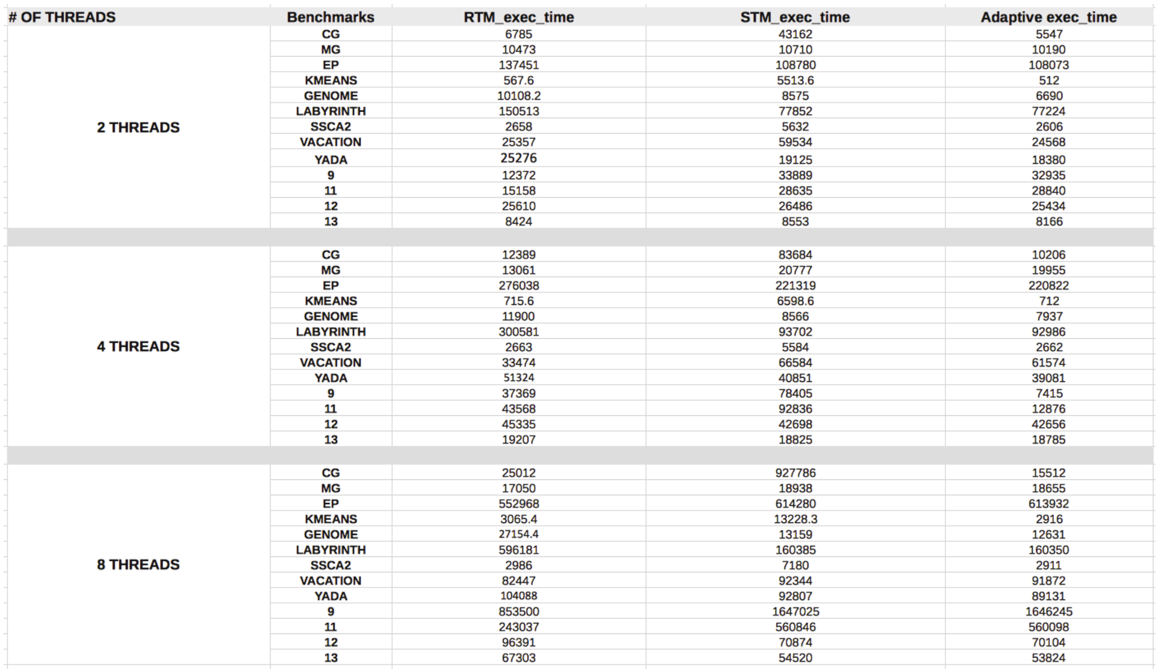
Benchmarks	PC_TXN	STM Time(ms)	RTM Time(ms)	Read-set_Size	Write-set_Size	Txn_Size	Write_Ratio	D.T Prediction	Optimum System
KMEANS	402748	12881	2673	15400960	52363264	300	0.7727272727	RTM	RTM
	4028b2	192	0	1026715	1026715	117	0.5		//
	402956	0	0	376	376	103	0.5		//
GENOME	402986	8815	24503	41992177	32652	259	0.0007819869	STM	STM
	402c60	2	1	21728	16321	116	0.3440708338	RTM	RTM
	402e05	4315	3407	40543510	2057244	536	0.0487487928	STM	RTM
	402f3e	4	2	52050	32642	133	0.3860354554	RTM	RTM
	40321d	8	1	107612	81600	154	0.431388096	RTM	RTM
LABYRINTH	403d7e	0	0	4120	512	134	0.1105354059		//
	403ec8	160384	596181	115260	181378	214	0.6114456004	STM	STM
	403e59	0	0	34	16	61	0.32		//
SSCA2	402acd	0	0	8	8	158	0.5		//
	403173	0	0	8	8	118	0.5		//
	403b0d	7180	2986	22362279	44724558	452	0.6666666667	RTM	RTM
VACATION	40187a	85142	74549	1506135671	22599539	576	0.0147831612	STM	RTM
	401e16	5352	5993	76715350	5797560	537	0.0702624595	STM	STM
	401ba2	1842	1898	25321248	5797560	155	0.1863040512	STM	STM
YADA	4053e6	529	115	2538569	1224457	101	0.3259014357	RTM	RTM
	4054c7	834	48	581558	0	115	0	RTM	RTM
	405571	91087	103812	104568795	24242674	626	0.1897587337	STM	STM
	405619	80	24	0	481786	95	1	RTM	RTM
	4056c3	263	70	1131533	507161	109	0.3094558382	RTM	RTM
CG	402add	7	28	0	0	146	0	RTM	STM
	402d53	120736	19417	2267582	2978034	580	0.5677186435	RTM	RTM
	402fa1	114	2230	0	1133791	254	1	STM	STM
	403259	20	2	28000	28000	206	0.5	STM	RTM
	4033d5	12	39	2	2	138	0.5	RTM	STM
	4037c8	271	3293	0	0	157	0	STM	STM
MG	403f65	120	60	64	64	130	0.5	RTM	RTM
	404554	18818	16990	8008	8008	276	0.5	STM	RTM
EP	401d4f	0	0	16	16	228	0.5		//
	4020d4	580309	282834	8192	8192	463	0.5	STM	RTM
	4022a1	33966	270128	16384	16384	701	0.5	STM	STM
DPOP_09	40164b	1647025	25350	225000	225000	241	0.5	STM	RTM
DPOP_11	401b11	560846	43037	116244833	0	210	0	STM	RTM
DPOP_12	401754	70874	96391	16	48024	821	0.9996669442	STM	STM
DPOP_13	401bae	54520	67303	8	8	598	0.5	STM	STM

C. Performance Comparison



# OF THREADS	Benchmarks	RTM_ENERGY (mJ)	STM_ENERGY (mJ)	ADAPTIVE
2 THREADS	CG	108.577	430.349	132.965
	MG	2532.616	2637.155	2548.272
	EP	732.396	1451.93	1451.93
	KMEANS	82.306	152.69	82.306
	GENOME	115.455	139.757	138.785
	LABYRINTH	1131.066	1104.958	1104.958
	SSCA2	330.914	410.457	410.457
	VACATION	328.349	930.438	328.349
	YADA	357.095	496.628	434.355
	9	17.418	502.302	502.302
	11	55.719	323.983	323.983
	12	255.816	404.723	404.723
	13	69.657	95.856	95.856
4 THREADS	CG	139.76	610.232	288.46
	MG	2541.445	2800.942	2708.95
	EP	761.633	2517.247	2517.247
	KMEANS	87.711	155.589	87.711
	GENOME	108.246	121.316	119.224
	LABYRINTH	1142.668	1040.032	1040.032
	SSCA2	506.169	525.457	506.169
	VACATION	325.856	814.888	723.502
	YADA	380.713	501.576	498.483
	9	27.15	2831.791	2831.791
	11	92.438	1711.879	1711.879
	12	257.39	607.317	607.317
	13	111.879	322.34	322.34
8 THREADS	CG	162.72	5627.366	354.855
	MG	2552.219	2812.593	2721.065
	EP	792.524	4562.191	4562.191
	KMEANS	102.069	164.942	102.069
	GENOME	111.632	118.371	115.734
	LABYRINTH	1163.222	1056.812	1056.812
	SSCA2	681.505	684.396	681.505
	VACATION	264.355	718.179	718.179
	YADA	380.713	504.042	468.281
	9	54.436	11706.527	11706.527
	11	166.735	4086.815	4086.815
	12	259.891	821.667	821.667
	13	200.244	697.366	697.366

D. Energy Expenditure Comparison



# OF THREADS	Benchmarks	RTM_exec_time	STM_exec_time	Adaptive exec_time
2 THREADS	CG	6785	43162	5547
	MG	10473	10710	10190
	EP	137451	108780	108073
	KMEANS	567.6	5513.6	512
	GENOME	10108.2	8575	6690
	LABYRINTH	150513	77852	77224
	SSCA2	2658	5632	2606
	VACATION	25357	59534	24568
	YADA	25276	19125	18380
	9	12372	33889	32935
	11	15158	28635	28840
	12	25610	26486	25434
	13	8424	8553	8166
4 THREADS	CG	12389	83684	10206
	MG	13061	20777	19955
	EP	276038	221319	220822
	KMEANS	715.6	6598.6	712
	GENOME	11900	8566	7937
	LABYRINTH	300581	93702	92986
	SSCA2	2663	5584	2662
	VACATION	33474	66584	61574
	YADA	51324	40851	39081
	9	37369	78405	7415
	11	43568	92836	12876
	12	45335	42698	42656
	13	19207	18825	18785
8 THREADS	CG	25012	927786	15512
	MG	17050	18938	18655
	EP	552968	614280	613932
	KMEANS	3065.4	13228.3	2916
	GENOME	27154.4	13159	12631
	LABYRINTH	596181	160385	160350
	SSCA2	2986	7180	2911
	VACATION	82447	92344	91872
	YADA	104088	92807	89131
	9	853500	1647025	1646245
	11	243037	560846	560098
	12	96391	70874	70104
	13	67303	54520	53824

Bibliography

[1] 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, New York, NY, USA, 2012, PACT '12, pp. 127–136.

[2] 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), Seattle, WA, USA, Sep. 2008, pp. 35–46.

[3] C. Wang, Y. Wu , et al. "Code generation and optimization for transactional memory constructs in an unmanaged language", In the Proceedings of the International Symposium on Code Generation and Optimization. IEEE Computer Society, 2007, pp. 34-48.

[4] 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.

[5] D. Christie , J. Chung , S. Diestelhorst , M. Hohmuth , M. Pohlack and et al. "Evaluation of AMD's advanced synchronization facility within a complete transactional memory stack", Proceedings of the 5th European conference on Computer systems, Paris, April 13-16, 2010.

[6] D. Dice, O. Shalev, and N. Shavit, "Transactional Locking II", In Proceedings of the 20th International Symposium on Distributed Computing, September 2006, pp. 194-208,

[7] D. Didona, P. Felber, D. Harmanci, P. Romano, J. Schenker, "Identifying the Optimal Level of Parallelism in Transactional Memory Applications", NETYS, 2013, pp. 233-247.

[8] D. Patterson, "Origins and Vision of the UC Berkeley Parallel Computing Laboratory", Chapter 1, 2004.

[9] F. Martinez and E. Ipek, "Dynamic Multicore Resource Management: A Machine Learning Approach", IEEE Micro 29, September 2009, pp. 8-17.

[10] G. Cunha, 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, 2008, pp. 251–256.

[11] G. E. Moore., "Cramming More Components onto Integrated Circuits", Textbook- Electronics, pp:114–117, April 1965.

[12] H.Tim, J. Larus, and R. Rajwar. Tech: "Transactional Memory", Morgan & Claypool, 2010.

[13] IBM, "Sabre Transactional Processing", Web link: http://www03.ibm.com/ibm/history/ibm100/us/en/icons/sabre

[14] I. Calciu , J. Gottschlich , T. Shpeisman , G. Pokam , M. 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.

[15] 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

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

[17] Intel Corporation, "Intel Architecture instruction set extensions programming reference" Website link: http://software.intel.com/sites/default/files/69/60/41604

[18] J. Bennett, J. Carter, and W.Zwaenepoel, "Distributed shared memory based on type-specific memory coherence". Vol. 25. No. 3, 1990.

[19] J. David, and J. Wendt, "Computational fluid dynamics", Vol. 206, New York, McGraw-Hill, 1995.

[20] J. El-Rewini, H. and M. Abd-El-Barr, "Shared Memory Architecture, in Advanced Computer Architecture and Parallel Processing", John Wiley & Sons, Inc., Hoboken, NJ, 2004.

[21] J. Gottschlich, M. Vachharajani, and J. Siek, "An efficient software transactional memory using commit-time invalidation", In Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization (CGO '10). ACM, New York, NY, 2010, pp. 101-110.

[22] J. Quinlan, " Induction of Decision Tree. Machine learning", 1986, pp. 81-106.

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

[24] J. Shewchuk and J. Richard, "Delaunay Refinement Algorithms for Triangular Mesh Generation", 2001.

[25] M. Bohlin, Y. Lu, J. Kraft, P. Kreuger, and T. Nolte, "Best-effort simulation-based timing analysis using hill-climbing with random restarts," Mälardalen University, Technical Report ISSN 1404-3041 ISRN MDH-MRTC-236/2009-1-SE, June 2009

[26] M. Castro, L. Góes, L. Fernandes, J. Méhaut, "Dynamic Thread Mapping Based on Machine Learning for Transactional Memory Applications", Euro-Par, 2012, pp. 465-476.

[27] M. Herlihy and J. E. B. Moss, Transactional memory: Architectural support for lock-free data structures, in Proceedings of the Twentieth Annual International Symposium on Computer Architecture (ISCA) , May 1993, pp. 289–300.

[28] M. Pereira, M. Gaudet, J. Amaral, and G. Araújo, "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), 2014.

[29] M. 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.

[30] N. Shavit and D. Touitou, "Software transactional memory", Distributed Computing, 1997, pp.99-116.

[31] 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.

[32] P. McKenney, M. Gupta, M. Michael, P. Howard, J. Triplett, and J. Walpole, "Is parallel programming hard, and if so, why Control", 2002.

[33] S. Chaudhry, R. Cypher, M. Ekman, M. Karlsson, et al. , "Simultaneous speculative threading: a novel pipeline architecture implemented in sun's rock processor", SIGARCH Comput. Archit. News 37, June 2009, pp. 484-495.

[34] S. Ramos and T. Hoefler,"Modeling communication in cache-coherent SMP systems: a case-study with Xeon Phi." In Proceedings of the 22nd international symposium on High-performance parallel and distributed computing (HPDC '13), New York, NY, pp. 97-108.

[35] Y. Rughetti et al.,"Automatic Tuning of the Parallelism Degree in Hardware Transactional Memory", Euro-Par 2014 Parallel Processing, Springer International Publishing, 2014, pp. 475–486.

[36] Y. Xiao et. al, "Automatic Optimization of Software Transactional Memory through Linear Regression and Decision Tree", to appear in the International Conference on Algorithms and Architectures for Parallel Processing, 2015.

[37] Z. Li, A. Jannesari, F. 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, October 2013, pp. 1004-1013.