2015

Optimization of Software Transactional Memory through Linear Regression and Decision Tree

Xiao, Yang

Optimization of Software Transactional Memory through Linear Regression and Decision Tree

by

Yang Xiao

A thesis
presented to Lakehead University
in fulfillment of the thesis requirement
for the degree of Masters of Science
in Electrical & Computer Engineering

Thunder Bay, Ontario, Canada
September, 2015

Abstract

Software Transactional Memory (STM) is a promising paradigm that facilitates programming for shared memory multiprocessors. In STM programs, synchronization of accesses to the shared memory locations is fully handled by STM library and does not require any intervention by programmers. While STM eases parallel programming, it results in run-time overhead which increases execution time of certain applications. In this thesis, we focus on overhead of STM and propose optimization techniques to enhance speed of STM applications. In particular, we focus on size of transaction, read-set, and write-set and show that execution time of applications significantly changes by varying these parameters. Optimizing these parameters manually is a time consuming process and requires significant labor work. We exploit Linear Regression (LR) and propose an optimization technique that decides on these parameters automatically. We further enhance this technique by using decision tree. The decision tree improves accuracy of predictions by selecting appropriate LR model for a given transaction. We evaluate our optimization techniques using a set of benchmarks from Stamp, NAS and DiscoPoP benchmark suites. Our experimental results reveal that LR and decision tree together are able to improve performance of STM programs up to 54.8%.

Acknowledgements

Firstly, I would like to express my sincere gratitude to my advisor Prof. E. Atoofian for the continuous support of my Master study and related research, for his patience, motivation, and immense knowledge. His guidance helped me in all the time of research and writing of this thesis.

Besides my advisor, I would like to thank the rest of my thesis committee: Prof. A. Jannesari, Prof. Luara, for their insightful comments and encouragement, but also for the hard question which incented me to widen my research from various perspectives.

My sincere thanks also goes to my classmates: Ahsan and Thireshan as well as my father and mother.

Table of Contents

List of Figures

List of Tables

List of Symbols

Symbol	Meaning
α	Weight variable of adaptive transaction scheduling
CI	Contention intensity
CC	Current contention
y	Output of linear regression
B	Intercept
x	Input of linear regression
ε	Error of linear regression
Tx_p	Prediction transaction size
TxSize	Transaction size
RdSize	Read-set size
WrSize	Write-set size
TS	Predicted transaction size
ST	Transaction size
WS	Write-set size
RS	Read-set size
SNT	Size of next transaction
NCT	number of assembly instructions between two consecutive transactions
WN	write-set of the next transaction
RN	read-set of the next transaction
TL	number of assembly instructions in a loop

List of Abbreviations

Abbreviation	Meaning
STM	Software Transactional Memory
LR	Linear Regression
TM	Transactional Memory
HTM	Hardware Transactional Memory
SVM	Support vector machine
TX	Transaction
TL2	Transactional Locking II
RV	Read version
TLC	Thread local clock
ProPS	Progressively Pessimistic scheduler
CL	Concurrency level
ATS	Adaptive transaction scheduling
STAMP	Stanford Transactional Applications for Multi-Processing
ID3	Iterative Dichotomiser 3
Adaboost	Adaptive boosting
API	Application Programming Interface
DP	DiscoPoP
NAS	NAS benchmark suit
OS	Operating system

Chapter 1

Introduction

Transistor scaling was the driving force for rapid growth of general-purpose processors in the past decades. Advances in integrated circuit technology allow processor designers to exploit faster and smaller transistors and boost performance of processors. The unprecedented growth in performance of processors enabled programmers to rely on hardware to increase the speed of their applications; the same software runs faster as chip manufacturers introduce new generations of processors. However, this trend has changed since 2003 due to energy consumption and heat dissipation issues that limited frequency scaling in single core processors. Since then, all major chip manufacturers such as Intel, AMD, and IBM turned in to multi-core processors to increase computational power of general-purpose processors. This shift in the landscape of general-purpose processors had tremendous impact on software developer community.

Traditionally, the vast majority of programmers developed sequential programs for single core processors. The programmers have become accustomed to the expectation that their programs run faster with each new generation of processors. However, this expectation is not valid in the era of multi-core processors. A sequential program runs only on one of the cores in a multi-core processor which is not significantly faster than single core processors. The only way that programs can continue to enjoy performance improvement in each generation of multi-core processors is parallel programming.

Parallel programming is a method to separate a large task into smaller sub-tasks which are then mapped into threads and are executed simultaneously. Compared with sequential programming, parallel programming can really reflect the benefit of multi-core processors by exploiting thread level parallelism in addition to instruction level parallelism. The conventional method of parallel programming is lock where

shared variables are surrounded by locks to guarantee atomicity of accesses to the shared variables. However, lock-based programming is challenging as it may lead to tricky synchronization bugs such as deadlock, livelock, etc [2]. To make parallel programming mainstream, it is necessary to find new programming models which simplify parallel programming for average programmers.

An alternative approach to lock-based programming is Transactional Memory (TM) [1]. TM is a programming model which facilitates parallel programming for multi-core processors. TM provides an atomic construct, called transaction, which is used to protect shared memory locations from concurrent accesses by threads. Reads and writes to transactional data occur at a single instance of time. Intermediate transactional values are not visible to other transactions. TM executes transactions speculatively in parallel and monitor memory locations accessed by active transactions. If executing transactions do not conflict over shared memory locations, then they safely commit. However, in the event of conflict, only one transaction can proceed and the rest should abort and restart. Transactions log operations during the execution so that they can restore state of the running program if roll-back is needed.

TM eliminates many of the problems associated with locks and enables programmers to compose scalable applications safely. In a TM program, a programmer does not need to worry about priority inversion, deadlock, or live lock. This is in contrast to lock-based programming in which a programmer needs to deal with lock placement and synchronization bugs. In a TM program, the programmer only needs to reason locally about shared memory locations and mark sections of the program that should be executed concurrently. The underlying system guarantees correctness. In addition to ease of programming, TMs are speculative in nature. The benefit of speculative approach is that transactions do not need to wait for shared memory locations; instead, they can execute concurrently and modify disjoint memory locations safely, leading to performance gains.

Transactional memory may be implemented in hardware (HTM) [1], software (STM) [2], or a combination of the two [3]. While HTM makes transactional memory fast, it increases design complexity and is not flexible. In addition, both HTM and hybrid approaches require adding new features to the hardware. STM, however, can use available features of current processors and comes with fewer intrinsic limitations imposed by hardware structures, such as buffer size and caches.

1.1 Software Transactional Memory

In the last decade, there have been several implementations of STMs [4, 5, 6]. The emergence of new STM algorithms has not been slowed down in the recent years, and the support for transactional memory in new processors [7] is likely to increase the number of STM implementations. The performance of STMs depends on several factors such as lock acquisition time, granularity of conflict detection, the mapping of memory addresses to the lock table, etc. Some researchers have explored design space of STMs and proposed changing STM parameters during the run-time. For example, Marathe et al. [8] studied lock acquisition in STMs and showed that the time at which locks are acquired has drastic impact on scalability. While eager policy (encounter-time locking) reduces overhead, lazy policy (commit-time locking) provides better throughput for some multithreaded applications. Marathe et al. [8] proposed an adaptive technique which dynamically changes lock acquisition policy in run-time. The other example is granularity of conflict detection [9]. Felber et al. [9] showed that performance of STMs varies with granularity of conflict detection and non-optimum parameters can slow down some programs by a factor of three. In addition, several STM implementations have partial roll-back ability. This ability can keep the validated part of a transaction and just retry the in-validated part. While the above techniques improve performance of STMs, all of them focus on execution of STM programs during the run-time. They do not provide any guidelines for programmers to write an efficient TM program in the first place.

1.2 Factors that Impact Performance of STMs

The size of a transaction has significant impact on performance. If the transaction is too short, then the overhead of STM APIs exceeds performance gain of parallel execution and may lead to an STM program which is slower than sequential version of the program. On the other side, if a transaction is too large, then the cost of roll-back in applications with high abort rate may reduce speed-up in STM applications.

Size of transaction is not the only factor that impacts performance of STMs. The other factors that affect execution time of transactions are read-set and write-set. When a transaction commits, all shared variables in the read-set and the write-set need to be checked and validated. If checking or validation fails, then the transaction needs to abort and retry. Transactions with large read-sets and write-sets are more likely to abort as there are more shared variables in large read-sets and write-sets which increase the probability of validation failure.

1.2.1 Brief introduction of contribution

One way to find optimal sizes for transaction, read-set and write-set is using try and error approach. A programmer can vary a transaction and finds out the optimal transaction size, read-set size as well as write-set size by running the program multiple times. This procedure is very time consuming and requires significant programming effort. To address this challenge, we propose two optimization techniques that automatically determine near optimal transaction size: the first technique exploits Linear Regression (LR) [10] to predict transaction size. LR receives parameters of a non-optimized transaction such as transaction size, read-set size, and write-set size and predicts the optimum transaction size. While LR is simple to implement, as we will show later, its accuracy is low. Our second optimization technique exploits a classifier and enhances accuracy of predictions. The classifier divides transactions into multiple groups and then uses a different LR model for each group. We also evaluated three different classifiers: decision tree, SVM and adaboost

decision tree. Our evaluations show that the accuracy of decision tree is higher than the other two as decision tree is more resilient to noisy dataset. Using a set of benchmarks from NAS [11], DiscoPoP [12], and Stamp [18] benchmark suites, we show that decision tree and LR together increase performance significantly.

1.3 Organization of the Thesis

The rest of this thesis is organized as follows. In chapter 2, we explain the necessary background for our optimization techniques and review related work. Chapter 3 discusses our optimization techniques in details. Chapter 4 reports experimental results. Finally, in chapter 5, we offer concluding remarks and future work.

Chapter 2

Background and Related Work

In this chapter, we review research papers that are related to this thesis. In Section 2.1, we explain TL2 which is an STM library and is used to evaluate our optimization techniques. In Section 2.2, we review research papers related to optimization of STMs. In Section 2.3, we explain linear regression which is used to predict transactional parameters. In Section 2.4, we discuss different types of classifiers that we used to categorize STM applications. Finally, in Section 2.5, we discuss NAS benchmark suite used in this thesis.

2.1 Transaction Locking II (TL2) [4]

TM is an optimistic approach and executes transactions speculatively. If transactions conflict then they abort and restart. On the other side, in lock-based parallel programs, threads conservatively acquire locks. This may serialize execution of threads unnecessarily and hurt performance. Figure 2-1 shows three threads executing six transactions. Executions of TX1, TX3, and TX5 overlap. These transactions can commit if they do not conflict. However, in lock-based programs, always critical sections are serialized. This reduces thread level parallelism and degrades performance.

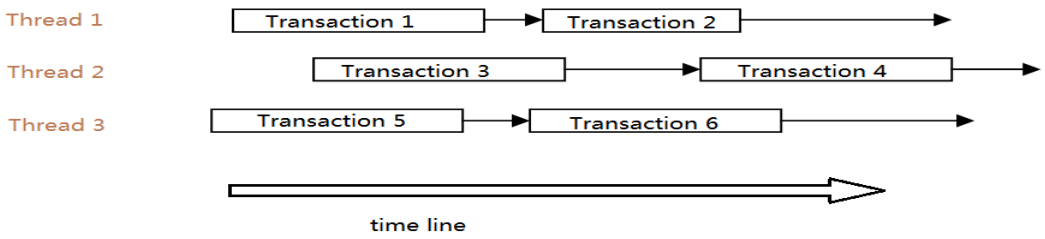

Figure 2-1: Transaction execution in STM.

In this thesis, we use TL2 [4] which is a popular implementation of software transactional memory and is faster than parallel programs written in pthread [13] up to 6X [4]. TL2 uses a global clock and a lock table to maintain consistency of

transactions. The global clock is a shared counter and is incremented by committing transactions. The lock table consists of a table of locks. Addresses of shared variables are hashed into the table entries and each entry of the table has two fields: lock bit and version number. The lock bit shows whether the corresponding variable is acquired or it is free. The version number is equal to the value of the global clock at the time that the last writing transaction successfully wrote into the corresponding variable.

When a transaction starts, it samples the value of the global clock and writes it into a local variable called read version (RV). Each transaction in TL2 keeps a read-set and a write-set which are linked-lists and store information related to read and written variables, respectively. Before a transactions commits, it starts validation of its read-set. To validate a variable, TL2 checks that lock bit of the corresponding lock entry is free. TL2 also compares version number of corresponding lock entry with rv. If the version number is less than or equal to RV, validation passes; otherwise, validation fails since another transaction wrote into the same variable and committed.

After validation of read-set, TL2 processes its write-set. If a lock bit of a variable in the write-set is free, then the transaction tries to acquire the lock bit. If lock acquisition fails, transaction aborts and restarts. Finally, transaction re-validates its read-set to make sure that it is not changed since last validation. Transaction can commit only if read-set validation passes and it successfully acquires locks for its write-set; otherwise, it aborts and restarts. Figure 2-2 shows the steps taken by a transaction to commit.

TL2 also implements a high efficiency read-only transaction validation process. Read-only transactions are those transactions that do not have any node in write-set. Therefore, it is not required to acquire locks for read-only transactions. In TL2, read-only transactions only need to do a post-validation to guarantee shared variables are consistent. If the post-validation fails, then the transaction is abord.

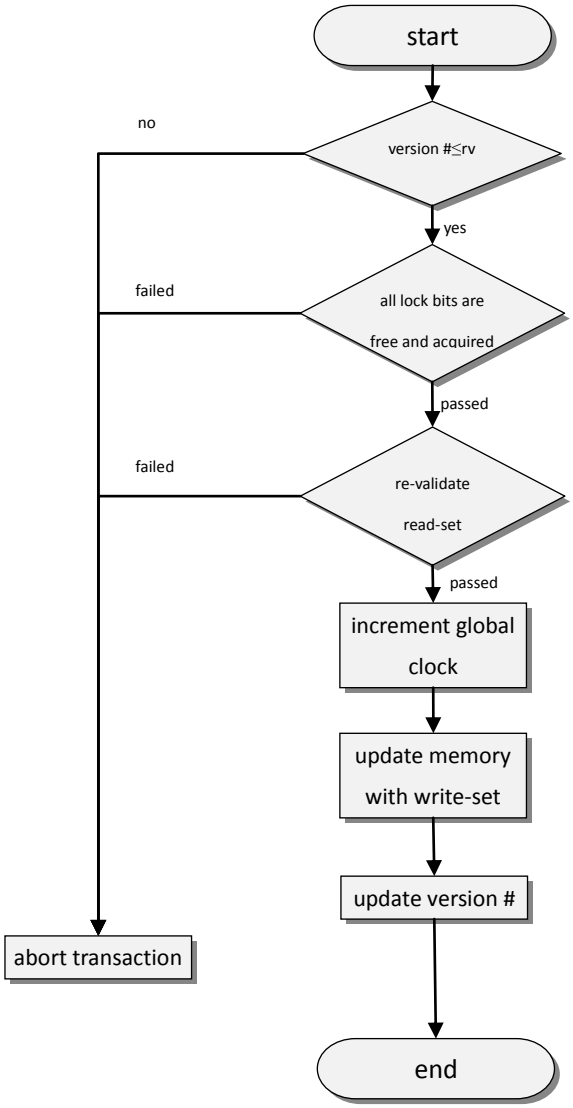

Figure 2-2: Steps of commit in TL2.

2.2 Optimization Techniques for STMs

Despite of ease of programming, STM has its own disadvantages. For example, the global clock is a bottleneck in STM as it is shared and modified by all writing transactions. In addition, in the event of conflict, only one transaction can proceed and the rest should abort. This increases program execution time and wastes processor resources. Therefore, there have been many research papers that try to optimize transactional memory.

One approach to optimize STM is to design and implement a new STM library from scratch. An example of this approach is TinySTM [6] which is a lighter and a faster implementation of STM than TL2. Although, this method may result in significant improvement, it is a very time consuming process and requires a relatively high level of knowledge on computer architecture, programming languages, etc.

The other approach is to optimize an existing implementation of transactional memory. In this approach, a researcher only focuses on those aspects of TM that require optimization and replaces/modifies them with new optimization techniques. In this Section, we focus on research papers that use the latter approach.

Partial rollback is a technique to reduce overhead of aborts in STMs [14, 15, 16]. Many existing transactional memory libraries abort the whole transaction if validation of read-set or acquisition of lock bits fails. However, sometimes, part of the aborted transaction is still correct. If we keep those correct parts and re-execute the reset, then we can save time theoretically.

Porfirio et al. [14] implemented the partial rollback technique in TinySTM. This work uses snapshot extension to determine the parts that need to be aborted. The evaluation shows that partial rollback quite often has better performance than baseline TinySTM.

However, in some benchmarks, it increases execution time. The main reason for slow-down is overhead. In benchmarks with high conflicts and short transactions, the partial rollback scheme needs to check validation of shared variables whenever conflict occurs. The execution time of partial rollback validation checking can take a large portion of total execution time. Therefore, the amount of time saved by partial rollback may be less than its overhead. The other reason for slow down of this technique is related to the behaviour of some of the transactions. In some transactions, the correct part is only a small fraction of total transaction, which means the benefit of correct part is small. Therefore, it is hard to gain speed-up in these types of transactions.

Global clock is s shared variable and is accessed by all writing transactions. This may result in ping-pong effect [17] and severally degrade performance. To cope with this problem, Avni et al. [17] proposed thread local clock (TLC) which replaces global clock with lock clocks.

In TLC, each thread has a local clock which is initialized to zero and is incremented by one at the start of every new transaction. There is also a thread local array that has an entry per thread recording timestamp of the thread. Each lock entry has a new field which is ID of the last writing thread. When a transaction commits it writes its thread ID and timestamp into the associated lock.

To validate read-set, all locks corresponding to the transactional read operations are checked to be unlocked. Then, the timestamp of each lock is checked to make sure that it is less than the associated thread j's entry in the thread local array. If the check fails then thread j's entry in the array is updated with the new timestamp.

If validation of read-set passes, TLC acquires lock bits of its write-set (similar to TL2 [4]). Then, TLC revalidates its read-set. If committing transaction successfully validates its read-set and acquires locks for its write-set, it increments its local clock and uses it to update version number of lock entries corresponding to its write-set; otherwise, it aborts and restarts.

While TLC eliminates central global clock, it increases abort rate since the new timestamp of a committed transaction is not transferred to other transactions immediately. Instead, other transactions notice the new timestamp when their validations fail. As such, TLC may degrade performance despite of the fact that it eliminates the central clock. In addition, Avni and Shavit evaluated TLC with micro benchmarks which are not representative of real applications. In contrast, we have evaluated our optimization techniques with the comprehensive Stamp [18] and NAS [11] benchmark suites.

Both TLC and partial rollback are dynamic approaches and result in runtime overhead. On the other side, our optimization techniques are static and do not incur any timing overhead.

Felber et al. [9] proposed a self-tuning methodology which dynamically adjusts concurrency level in STMs. One of the key factors in STM programs is contention. Too many threads in a program increase contentions over shared memory locations and hurt performance. On the other side, if concurrency level is too low, then exploited parallelism by STM programs will be limited. The optimum number of executing threads depends on many parameters including but not limited to pattern of addresses generated by transactions, OS scheduler, structure of memory hierarchy, etc. So, identifying the right level of concurrency in STMs is not a trivial task. Felber et al. [9] used a hill-climbing algorithm to explore concurrency level space in shared memory STMs.

One of the shortcomings of this work is response time. In some benchmarks, a transaction commits before the dynamic approach finds the best concurrency level. For these benchmarks, the response time is too long to result in any noticeable speed-up. On the other side, our optimization techniques are applied before runtime. Hence, response time is not an issue in our work.

Wang et al. [19] developed a compiler that automatically optimizes programs written in C/C++. The compiler focuses on synchronization barriers and tries to remove those barriers that are not necessary for correctness of parallel programs. Synchronization barriers are used to maintain consistency but reduce the concurrency level. However, non-experienced programmers may use a conservative approach and add redundant synchronization barriers to guarantee correctness of parallel programs. Redundant barriers reduce thread level parallelism and degrade performance. To remove redundant barriers, the complier checks the dependency of transactions. There are two situations that the compiler can remove a barrier. First, there is no dependence between two transactions. Secondly, there is only write-after-write dependence.

Write-after-write dependence can be checked at commit stage of STMs and does not require synchronization barriers. The compiler improves performance of most of parallel benchmarks considerably. However, some benchmarks only show negligible improvement because transactions in these benchmarks are large and quite often conflict with each other.

To reduce contention in parallel applications, it is important to determine dependency between variables. DiscoPoP [12] is a tool that automatically finds parallelizable regions of a sequential code based on dependency of variables. DiscoPoP is able to identify parallelism between code regions with arbitrary granularity and does not require any predefined notion of language constructs. DiscoPoP identifies sections of the code in which data dependency does not exist. These sections are called Computational Units (CUs). Then, the tool builds a dependency graph using CUs. Nodes of the graph represent CUs and edges represent dependency between CUs. From the dependency graph, DiscoPoP determines potential parallelism available on varying levels of the code. The output of the DiscoPoP is a file that indicates which lines of the sequential code can be grouped as a task and run concurrently with others. We used the set of benchmarks introduced in DiscoPoP for evaluation of our optimization techniques.

As mentioned earlier, one of the sources of overhead in STMs is contention. Rito et al. [20] proposed Progressively Pessimistic scheduler (ProPS) which is a scheduler for reducing contentions in STMs. ProPS exploits a matrix to indicate the concurrency level (CL). The rows and columns of this matrix are atomic operations. CL_{ij} indicates how many transactions executing atomic operations of type i may execute concurrently with one transaction executing atomic operation of type j. The scheduler adjusts the values in the matrix based on abort rate. If the scheduler notices that transactions frequently have conflicts, the scheduler decreases the corresponding values of the matrix. This scheduler uses the matrix to speculate conflict among executing transactions. The scheduler gives high priority to those threads which have

high values in the matrix. On the other side, the scheduler temporarily stalls or blocks those transactions that have low concurrency values in the matrix. The main benefit of ProPs is low overhead as it uses a matrix to maintain record of contentions in STM programs. As such, the response time of ProPs is low. However, simple and quick control function of this scheduler has a disappointing accuracy rate. In some benchmarks, the performance of STM with this scheduler is worse than the baseline STM. There are different sources that cause conflicts. For each source, we need to use an appropriate technique to adjust concurrency level. Blindly increasing or decreasing values in the matrix may lead to low accuracy rate. This can be explained through an example. Assume that a TM program has two threads: A and B. Thread A has three transactions and thread B has only one transaction. The first and third transactions in A conflict with the transaction in B but the second transaction in A has no conflict with the transaction in B. If we use ProPs in this example, the concurrency level is decreased after first conflict. Then, the second transaction will be blocked because of low concurrency level in the matrix. Finally, ProPs will increase concurrency level because the second transaction actually has no conflict, which leads to the third transaction in thread A conflict with the transaction in B. The accuracy rate in this example is 0. Our work is different as we use a static approach and optimize STM programs before the runtime.

Unlike ProPs, some research papers focused on scheduler using mathematical methods to reduce conflicts in STMs. Yoo et al. [21] proposed adaptive transaction scheduling (ATS) to adjust concurrency level. ATS uses equation 2-1 to quantify contention intensity:

$$CI_n=\alpha \times CI_{n-1} + (1-\alpha) \times CC \qquad (2\text{-}1)$$

Where CI_n is contention intensity in n^{th} execution of a transaction in a thread, CC is current contention, and α is weight variable. This equation is evaluated whenever a transaction commits or aborts. If a transaction commits CC is set to 0; otherwise, it is set to 1. Weight variable determines which part of the equation is more important, the past history or the current contention. Yoo and Lee [21] measured execution time

under different values for α and threshold and found that α = 0.3 and threshold = 0.5 result in the best average performance.

The scheduler uses a decentralized contention manager, which means each thread manages its own contention, locally. Before a TX starts, ATS uses equation 2-1 to determine CI. If CI is more than the threshold, ATS inserts the TX into a centralized queue. The structure of the queue is first-in-first-out (FIFO). ATS only allows the TX in the head of the queue to execute, effectively serializing transactions with high CI.

Mathias et al. [22] proposed a dynamic approach to tune important STM parameters such as different write strategies, hash-function for local write-set, etc. STM library samples some metrics such as the number of unique read and write locations, the number of aborts and commits, and the quality of hash functions to decide on STM parameters.

There are two approaches for adaptivity: local and global. In global adaptivity, STM parameters are changed for all running transactions. On the other side, in local adaptivity, STM parameters are changed on a per-thread basis. The main advantage of local adaptivity over global adaptivity is that every thread decides on STM parameters locally. This prevents costly synchronization operations among the executing threads. On the other side, global adaptivity is a bottleneck for scalability as it requires all threads in a TM program to be synchronized to change STM parameters. The disadvantage of local adaptivity is that global STM parameters such as hash function for lock table cannot be changed locally. This limits effectiveness of local adaptivity. To exploit the better of the two, Mathias et al. [22] used a hybrid scheme to change STM parameters. Evaluations with STAMP benchmarks reveal that the hybrid approach improves performance of the benchmarks by 10% on average

Our work is different from [21] and [22] as we optimize STM programs before the runtime. Our approach focuses on source code of STM programs and decides on some STM parameters such as TX size, read-set size, etc.

In conclusion, there are two common ways to optimize software transactional memory: optimizing the library of transactional memory and optimizing the source code of transactional memory programs. In this thesis, we focus on the latter technique and propose optimization techniques that require adjustments in the source code of the programs and not the STM libraries.

To optimize static parameters, we need to change the source code of programs. Generally speaking, many parameters such as size of transaction, read-set size, and write-set size can impact performance. We should not only consider the impact of individual parameters on performance. We also need to take into account the performance impact of these parameters together as some of these parameters are correlated.

One of the challenges of a static optimization technique is that it needs to explore a large space. The parameters that we focus on are continuous variables and so there are many combinations of those variable values that make it impossible to test them all manually. We need a tool that tests STM parameters and automatically optimizes STM programs.

One method to evaluate the impact of STM parameters is using a set of benchmarks. Application-based benchmarks are useful programs that help STM designers to explore design space of STMs. However, they have a limited ability to isolate the effect of each parameter on the overall performance. For example, an application's read-set size is often tied to the size of its transactions, but these two parameters may be completely orthogonal in terms of how they affect the system performance. To quantify the impact of STM parameters on performance, we need an evaluation framework which is able to isolate the impact of each parameter on performance.

EigenBench [23] is a micro-benchmark which can be used to fully evaluate STM systems. EigenBench decouples STM characteristics and enables programmers to vary each of those characteristics, independently. The characteristics considered in EigenBench are:

1. Concurrency: number of threads
2. Working-set size: size of read-set and write-set
3. Transaction length: size of transaction
4. Pollution: the percentage of shared write variables
5. Contention: the rate of conflicts
6. Temporal locality: probability of repeated addresses
7. Predominance: fraction of shared access cycles to total execution cycles
8. Density: the percentage of non-shared cycles executed outside of transactions

In this thesis, we consider only the first five parameters as the last three rely on memory access latency and processor cycles and so are not appropriate for a static optimization technique.

In EigenBench [23], a programmer can adjust 21 parameters such as number of threads, number of transactions per thread, etc. to change each of the eight character tics. Then, EigenBench [23] generates a program based on the selected values of the parameters. The program can be used to evaluate performance of one or more of those characteristics simultaneously.

We use EigenBench to evaluate the impact of transaction size, read-set size, etc on performance. Based on those evaluations, we can find the optimum values of transactional parameters. Then, we change source code of TM benchmarks based on the optimum values to gain speed-up. We will discuss details of our optimization technique in chapter 3.

Some research studies used neural network to optimize STMs. Neural network provides the ability to approximate different types of functions including functions with continuous variables. Inspired by the human brains, a neural network consists of a set of interconnected neurons which cooperate to compute a specific function. Neurons are the processing elements of a neural network and each neuron has a

simple function. In a neural network, each link is associated with a weight. The weight of a link determines the influence of neurons in a level on the next level neurons.

Rughetti et al. [27] proposed a self-adjusting concurrency scheme for STMs based on neural network. This self-adjusting concurrency scheme can activate or block threads to increase thread level parallelism and reduce data conflicts. This scheme has three parts: a collector, a neural network, and a controller. The collector monitors an application and collects a set of values characterizing the application. The set of values are passed to the controller after a sampling interval. The neural network receives the set of values from the controller and predicts the average wasted transaction execution time spent by the application. Then, the controller adjusts concurrency level according to prediction made by the neural network. The collector collects three parameters from a benchmark:

1. size of read-set
2. size of write-set
3. execution time of successfully committed transactions
4. execution time of non-transactional parts

Neural network in this scheme is a three layers radial basis function network. The first layer receives input parameters. The second layer calculates wasted time and sends it to the third layer which is output layer.

Rughetti et al. compared the neural network-based concurrency control scheme with TinySTM [6]. The performance of the baseline TinySTM is an increasing function of the number of thread. However, after certain point, it degrades due to data conflicts. On the other side, the performance of the enhanced scheme does not degrade as it adjusts number of threads to avoid data conflicts when the number of threads increases.

The main disadvantage of neural network is that there is no clear guideline for the structure of the neural network such as the number of layers, the weight of each connection, and the number of neurons in each layer. Each neural network should be built based on an application and there is no clear rule that we can follow. Unlike neural network, regression [10] is a statistical technique and has a clear approach to build regression model.

2.3 Linear Regression

Linear Regression (LR) is a mathematical equation which relates a response variable to a set of input parameters for a given design space [10]. LR is widely used to predict the response variable at an arbitrary point in the design space. Equation 2-2 shows a simple model for LR:

$$y = B_0 + \sum_{k=1}^{q} (B_k \times x_i) + \varepsilon \qquad (2\text{-}2)$$

Where y is response variable, x_i is an input parameter, B_0 is the intercept of the fit with the y-axis, and ε is the error of LR model. B_i ($0<i$) is coefficient and represents the expected change in y per unit change in x_i. LR uses least square method to find the best-fitting curve to a set of test points. In this method, coefficients are calculated so that the sum of square of the errors for the test points (error of a test point is the distance of the point from the fitting curve) is minimized. While LR exploits a simple model for prediction, it shows excellent results in many applications [28, 29, 30] and is able to predict the response variable with high accuracy.

Dong et al. [28] used linear regression to predict age of article writers. The linear regression model is based on the frequency of words in articles. Training and testing of datasets are from three corpora and forums: blog, fishing, and cancer. Dong et al. [28] selected articles from those corpora and calculate important features such as textual features and gender. Frequency of words varies from one corpus to the other. For example, people in fishing forum rarely talk about cancer while the word "cancer" is top high-frequency word in cancer forum. Therefore, it is important to distinguish

corpus-specific textual features. In addition, some features are commonly used in the three corpus such as "with", "and", "hence", etc. Dong et al. [28] built four types of regression models to predict age of an author:

1) A model trained by corpus-specific features individually

2) A model trained by all the three corpus-specific features

3) A model only trained by global features

4) A model trained by global features and individual corpus-specific features.

For prediction, the regression model analyzes a test article and measures the features. Then, the four models predict age of the article writer. The purpose of using four models for prediction is to find out which model has the highest accuracy. The evaluation shows that the correlation rate can reach up to 75% and error of prediction is 4.1 to 6.8 years. The model trained by all corpus-specific features has better performance than individual models and the model which only uses global features has the worst performance.

Google Inc. published [29] regression models to predict box office sales. The model inputs are based on phrases that clients search through Google search engine. The first model uses search volume to predict weekend box office sale. The accuracy of this model is 70%. The second model takes into account some extra factors such as number of theaters and franchise status to boost the accuracy rate to 92%. The third model is used to predict box office sales four weeks ahead. The accuracy of this model is 94%.

2.4 The Choice of a Classifier

A TM program is composed of a variety of transactions. These transactions vary in terms of TX size, read-set size, etc. Using a single linear regression model to predict parameters of all transaction types reduces accuracy (details will be discussed in chapter 3). The alternative approach is using multiple linear regression models. Each

model corresponds to a specific transaction type. To select which model should be used for a given TX, we need to use a classifier such as decision tree.

Classification is the task of assigning objects to a set of predefined categories. Decision tree [31] is a popular approach for classification. Originally, decision tree was used in the field of statistics. However, soon it found to be effective in many other disciplines such as machine learning, image processing, etc. A decision tree classifies an input object through a set of functions organized in a hieratical manner and represented by a tree. A tree has three types of nodes: root, internal, and leaf [31]. An internal node splits the objects into two categories according to a test function. The inputs to the function are attributes of the object and the output of the function is a binary value: 0 or 1. A leaf represents a category. Objects are classified by navigating them from the root down to the leaves, based on the output of the test functions along the path. There are many open source implementations of decision tree such as ID3 [25] and C4.5 [26].

Kemal et al. [32] proposed a hybrid intelligent method to improve classification accuracy for multi-class classification problems. The hybrid method is based on C4.5 decision tree and is evaluated using three multi-class problems: dermatology, image segmentation, and lymphography datasets. The accuracy of C4.5 for the three problems is 84%, 88%, and 80%, respectively. In Dermatology and lymphography, medical sciences overlap while in image segmentation, graphics overlap. Using single classifier, the overlaps create mutual interferences during the training and reduce accuracy. On the other side, the hybrid approach avoids overlaps and improves accuracy to 96%, 95%, and 87% in the three multi-class problems, respectively.

Support vector machine (SVM) is a machine learning method which is used to classify a set of data. SVM builds a model based on training data which can be used to classify test data. Similar to decision tree, SVM is a binary predictor and classifies objects into two categories.

SVM creates a hyperplane based on training data for classification. Figure 2-3 shows an example for SVM. There are two classes in the 2D-coordinate. There are many lines that separate the two classes. If a line is too close to an element, then the line is sensitive to noisy data. Therefore, the best line is the one which has the largest distance to the nearest training-data point of any class. Using a set of mathematical equations, SVM determines the hyperplane [33].

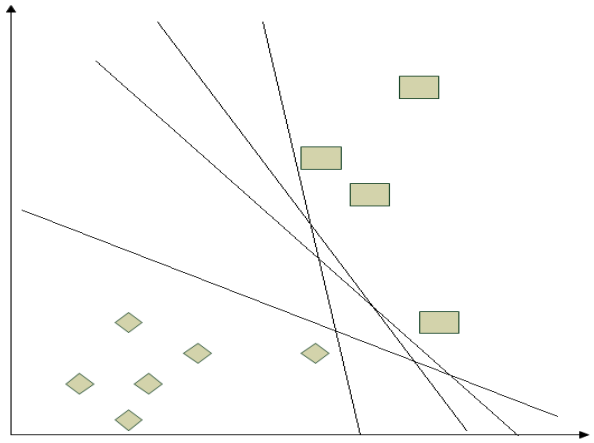

Figure 2-3: Two classes in a 2D-coordinate.

A natural advantage of decision tree is that it is simple to understand. Decision tree is similar to a white box that the decisions can be read and understood by human while SVM is similar to a black box model which is hard to understand and interpret as it relies on complicated mathematical equations. The other advantage of decision tree is that it is able to select only those features that correlate with the output and filter out irrelevant features. Furthermore, decision tree is insensitive to the noise when the training dataset is large because the large number of training datasets can dilute the influence of noise.

The other benefit of decision tree over SVM is related to scale problem. In machine learning, large values can mislead the training process because sometimes, large values can dominate small values. The good aspect of decision tree is that it does not suffer from scale problem. However, SVM which depends on mathematical equations need to deal with the scale problem. Last but not least, in both decision tree and SVM,

no linear relationship is required among features and output. On the other side, some techniques such as linear regression require linear relationship between features.

However, decision tree has its own disadvantages. The first one is that training dataset should be large; otherwise, it may lead to high error rate. Secondly, since each node in decision tree can have up to two children nodes, the size of the tree grows exponentially if we increase the number of decisions. On the other side, SVM can classify test data with high accuracy using a small set of training data.

In conclusion, decision tree is easy to understand and is suitable for applications with a few decision situations and with large amount of training dataset, whereas, SVM is suitable for medium sized training dataset.

Adaptive boosting (Adaboost) [34] is a machine learning technique which boosts performance of decision tree. Each sample in a machine learning technique may contain a large number of features. If we train the decision tree or SVM by all those features, the speed of training might be too low. Also, this may reduce the accuracy rate because of Hughes Effect [34]. Unlike basic decision tree or SVM, adaboost selects those features which improve the performance of predictions.

Adaboost iteratively trains classifiers using input datasets. Adaboost associates weights to inputs. The weight of those inputs that are misclassified is increased in each iteration. This biases classifiers towards correcting misclassified inputs in future iterations. The final result is an ensemble of instances of hard to classify inputs, each with its own weight. During the testing, unseen instances are classified using a weighted combination of weak classifiers.

Zhen et al. [35] used decision tree, SVM, and adaboost to select parallelizable sections of a sequential code. The inputs to the machine learning techniques are features of a snippet of a code and the output is a binary value: whether the snippet is parallelizable or not.

During the evaluation, the authors used two methods for training and testing. In the first method, all 16 features are used. The results show that adaboost has the best

performance which is 92%. The accuracy of decision tree and SVM is the same and is equal to 85%. In the second method, only top features with an importance score of 0.08 or greater are used. Accuracy of adaboost, decision tree, and SVM is 91%, 90%, and 89%, respectively.

Zhen et al. [35] concluded that the most important feature which increases accuracy is number of instructions whereas execution time is not a top feature. Among the three prediction methods, adaboost is more accurate than decision tree and SVM. The accuracy of decision tree and SVM increases when the numbers of features is narrowed down. Therefore, blindly increasing the number of features not only does not increase accuracy but also may reduce it.

Freund et al. [36] proposed an alternating decision tree based on adaboost decision tree. This alternating decision tree only focuses on binary decisions. An internal node in decision tree splits the objects into two categories according to a test function. Unlike the baseline decision tree, alternating decision tree [36] has two types of internal nodes: splitter nodes and prediction nodes. Splitter nodes have the same function as internal nodes of the baseline decision tree. They make decision according to a test function. Each prediction node is associated with a value. The leaf nodes in the baseline decision tree are used to make decision, whereas in the alternating decision tree, the final decision is the sum of all passed values of prediction nodes.

Compared with the baseline adaboost decision tree, adaboost alternating tree is more flexible. Each node in the baseline decision tree can split at most once but alternating tree does not have this restriction. Freund et al. [36] used Cleve dataset to evaluate adaboost alternating tree and showed that adaboost alternating tree requires only 6 nodes to represent Cleve dataset whereas the baseline decision tree requires 446 nodes. In addition, in the baseline decision tree, if we need to add new decisions, then we can only add them to the leaf nodes or rebuild the whole decision tree. However, in adaboost alternating tree, we can add new decisions to anywhere and only adjust the value of each node. Compared with the baseline decision tree model,

this paper [36] shows that the accuracy rate of adaboost alternating decision tree is 15% higher than the baseline decision tree.

2.5 NAS Benchmark Suite

To evaluate an STM system, researchers rely on a set of benchmarks. If the set of the benchmarks are selected from a specific field, then the outcome of the research is not reliable. To be able to extend the outcome of a research project to the real world applications, we need a set of comprehensive benchmarks that truly represent real world applications. Asanovic et al. [37] proposed 13 Dwarfs as a guideline to develop benchmark suites for parallel applications. A dwarf is a high level abstraction which categorizes applications based on patterns of computation and communication. Programs that belong to a pattern may have different implementations, but the underlying patterns do not change through different implementations. Their work is based on 7 dwarfs proposed by Phil Colella [38] who identified seven numerical methods which are important for science and engineering. Asanovic et al. [37] have examined different application domains, i.e. machine learning, computer games, etc and expanded the primary seven dwarfs to 13. Asanovic et al. [37] showed that NAS benchmark suite [11] includes all those dwarfs and so in this work, we use NAS benchmark suite to evaluate our optimization techniques.

Chapter 3

Static Optimization of Transactional Parameters

In this chapter, we focus on contributions of this thesis. In section 3.1, we explain how system APIs and the choice of programming style impact STM programs. In section 3.2, we describe parameters that impact performance of STMs. In section 3.3, we show how linear regression predicts a set of static parameters to improve performance. In section 3.4, we focus on classifiers and show how a classifier can assign an appropriate LR model to a benchmark. In section 3.5, we explain how a combination of linear regression and decision tree enhances performance of STM applications and finally, in section 3.6, we evaluate our optimization techniques for two and four threads.

3.1 System APIs and Programming Style in STMs

3.1.1 System APIs

In an STM program, the parallel parts are implemented in transactions. Although programmers do not need to worry about deadlock, live-lock, and other synchronization bugs, they still may get discouraged by performance of STMs.

In chapter 2, we discussed some parameters that impact performance of STMs. Long transactions may lead to high overhead when they abort and large write-sets and read-sets may cause high contentions. Therefore, if programmers are not careful about transaction, read-set, and write-set sizes, they can generate programs with inefficient structures leading to unacceptable performance.

To evaluate the impact of STM parameter on performance, we use DP benchmark suite [12]. DP is a set of sequential programs used to evaluate DiscoPoP (Section 2.2). DP is composed of 6 benchmarks:

- Histo_serial: It uses random numbers to generate histograms.
- Combined_ctrl_regions: This is a simulation of a controller to randomly mix colors based on three-primary colors.
- Ann_training: This benchmark is an implementation of neural network training

algorithm.

- Mandelbrot: This benchmark draws a picture of Mandelbrot set.
- Mc_light: This benchmark simulates the propagation of light using Monte Carlo approach.
- Mont: It is used to draw random curve.

DiscoPoP detects parallelizable sections of a sequential code and transforms the sequential code to a parallel code based on pthread library [13]. We converted the pthread codes generated by DiscoPoP to STM programs. The conversion is straightforward and requires replacement of pthread APIs with STM APIs. Lock/unlock in pthread is replaced by TM_BEGIN/TM_END and access to a shared variables in pthread is replaced by TM_SHARED_READ/TM_SHARED_WRITE.

In this thesis, we use two Intel Xeon E5660 processors to evaluate our optimization techniques. Each processor has six cores and is capable of running up to 12 threads simultaneously. Each processor has a 12MB shared L3 cache with 64B cache lines. Each core has a 32KB instruction cache and a 32KB data cache.

Figure 3-1 shows performance of STM relative to sequential programs in DP benchmark suite. Bars less than one show slow-down. For each benchmark, the number of threads varies from two to 8. While STM improves performance of some benchmarks such as *mandelbrot*, in some others, it degrades performance. In *mont*, execution time increases by a factor of 2 when the number of threads is 8.

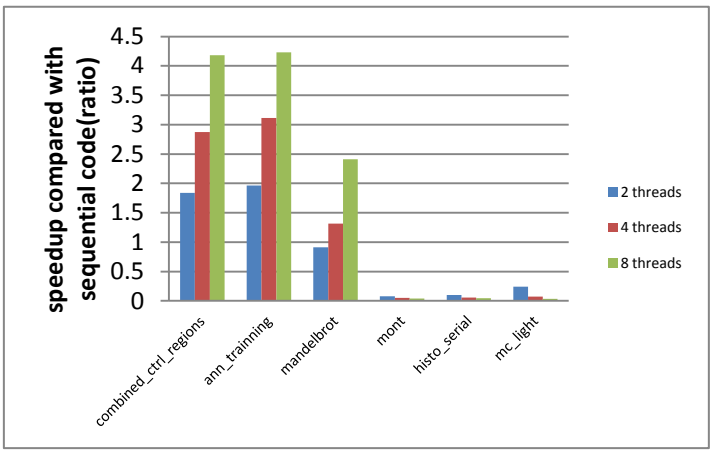

Figure 3-1: Performance of DP benchmarks in STM relative to sequential code. Bars less than one show slow down.

To scrutinize why STM degrades performance of some of the benchmarks, we focus on histo_serial benchmark. There are 4 transactions in this benchmark. We use *gettimeoftheday()* function to measure execution time of each transaction. Table 3-1 reports execution time per transaction. The first column indicates the transaction number. Other columns show execution time of transactions when the number of threads changes from 2 to 8. Execution time is averaged over 10 runs and is reported in Table 3-1.

Table 3-1: Execution time (in second) per transaction in histo_serial benchmark.

Transaction #	2 threads	4 threads	8 threads
1	5.87	9.64	14.96
2	0.07	0.04	0.04
3	0.29	0.22	0.17
4	3.11	5.94	5.79

From table 3-1, we observe that execution time of the first and the fourth transactions dominate the total execution time. Figures 3-2 and 3-3 show the code snippets of the first and the fourth transactions, respectively.

```
TM_BEGIN();
 for(i=lowIndex;i<highIndex;i++)
 {
    if(i<50)
       data[i]=(rand()%range_max)-range_min;
 }
TM_END();
```

Figure 3-2: The code snippet of the first transaction in histo_serial.

```
TM_BEGIN();
for(k=1;k<20;k++){
  /........
  /........
  TM_SHARED_WRITE_F(rhs[k], t);
  }

TM_END();
```

Figure 3-3: The code snippet of the fourth transaction in histo_serial.

The two transactions are small but they call external functions. The first transaction calls *rand()* to generate random values. This function is a system call function and is implemented in operating system (OS). The seed of this function is time which is provided by OS. When a system API is invoked, OS needs to fall into kernel mode. To do so, OS first stores context of current user process into memory and then switches to system API. This procedure is called context switching. If system call happens only once, the overhead of context switching is small. However, *rand()* in histo_serial benchmark is called 50 times in each transaction. This is the main reason that STM degrades performance of the first transaction.

To eliminate the overhead of system call, we wrote a pseudo-random number generator function and use it instead of *rand()*. The seed of this pseudo-random number generator is the number of loop iterations and a local snapshot of the global clock. This function keeps a dataset which stores a large number of values. Based on the seed, this function selects a value from the dataset. All the process is done by local statements to avoid context switching. Figure 3-4 shows execution time of the optimized histo_serial benchmark. In figure 3-4, *rand()* means the original version of histo_serial benchmark and *no_rand()* means the optimized version. Y-axis is execution time in second.

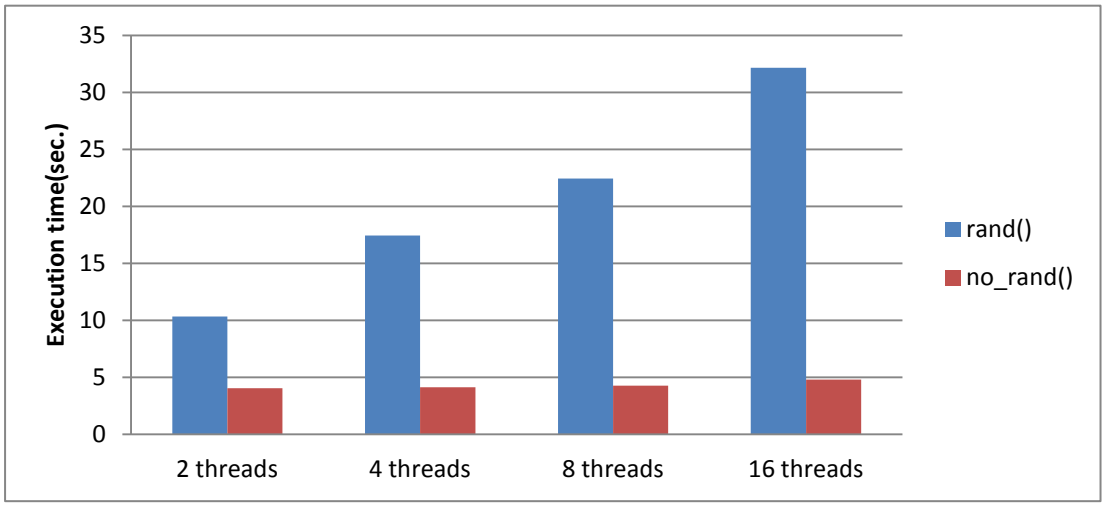

Figure 3-4: Execution time of histo_serial benchmark with *rand()* and without *rand()*.

This graph strongly proves our assumption. When using a local function, the speed-up is dramatically increased compared with *rand()* function. Therefore, in STM programs, we should avoid using external functions which can cause context switching.

The fourth transaction in *histo_serial* benchmark uses *printf()* function within a loop. This function is a standard IO function in C programming language. When a thread calls this function, system prints formatted data to a standard output which is usually a console or a terminal. This IO function generates a software interrupt. When OS receives an IO request through a software interrupt, it uses a system API to send output stream to a console or a terminal. At the same time, the calling thread is blocked until this IO process is finished. Essentially, *printf()* function is similar to *rand()* function and causes context switching.

Figure 3-5 compares performance of optimized and original histo_serial. In the optimized version, *printf()*function is removed. Y-axis is execution time in second. Performance of the optimized version is improved up to 30X.

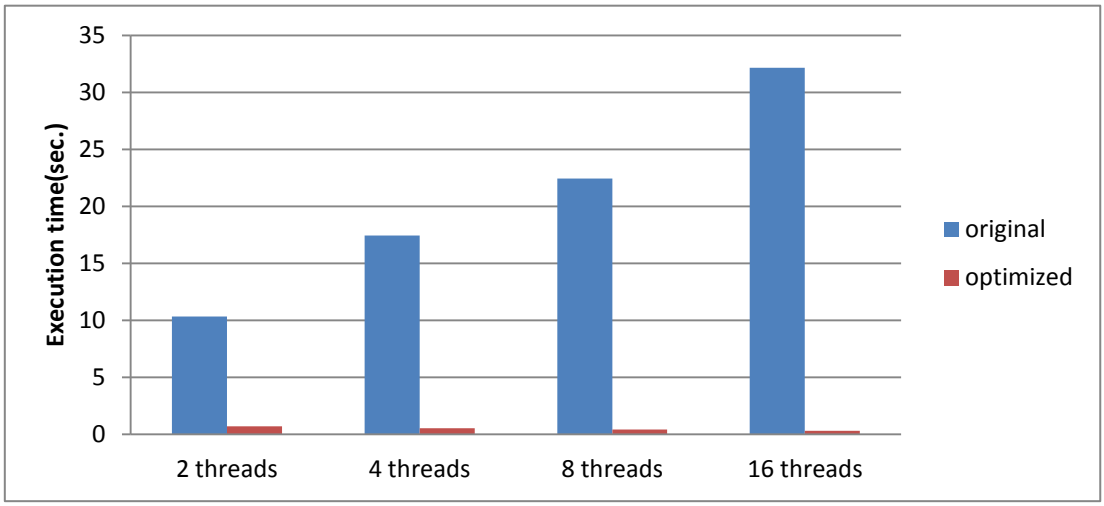

Figure 3-5: Execution time of optimized and original histo_serial.

We also investigate the other two benchmarks for slow-down: mont and mc_light. The problem in these two benchmarks is *rand()* function as well. After replacing *rand()* with pseudo-random number generator, performance of optimized benchmarks is improved dramatically. Table 3-2 exhibits the results. The numbers in the table are execution time in second.

Table 3-2: Execution time of each transaction in histo_serial benchmark.

Benchmark	2 threads	4 threads	8 threads
mont	21.26	33.56	45.21
mont_optimized	1.05	1.13	1.26
mc_light	2.12	7.39	16.54
mc_light_optimized	1.77	1.49	1.03

In a nutshell, context switching can dramatically decrease performance of STM programs and so should be avoided. Unfortunately, an optimizing compiler might not be able to remove them all because removing system calls may compromise correctness of programs. Therefore, what we can do is to notify programmers about potential context switching in transactions. Then, they can remove context switching by restructuring their programs.

3.1.2 Programming Style in STMs

In addition to context switching, coding style also affects the performance of STM programs. One of the popular data structures used in programs is array. Programs access elements of an array through indexing. If transactions of a STM program

access disjoint indexes of an array, then there is no conflict over the array. This can reduce overhead of the STM program as transactions do not need to call STM APIs such is STM_SHARED_READ and STM_SHARED_WRITE to access the array elements.

Figure 3-6A shows a sequential program that accesses an array in a loop. In each iteration, an element of the array is changed. Figure 3-5B shows STM version of the sequential program. Each transaction accesses a non-overlapping portion of the array. So, there is no need to use TM_SHARED_WRITE to access the array. However, an inexperienced programmer may consider this array as a shared variable and use STM APIs to guarantee consistency of the array.

```
for(j = 0; j<x; j++){
    counter[j]=3+j;
}
```

A

```
int numthread=getThreadNum();
int myId=getThreadId();
int low=(x*myId)/ numthread;
int high=(x*(myId+1))/numthread;
TM_BEGIN();
for(j = low; j<high; j++){
ct=3+j;
TM_SHARED_WRITE(counter[j],ct);
 }
TM_END();
```

B

Figure 3-6: A) A sequential program accesses an array. B) STM version of the program.

Three benchmarks in DP have structures similar to Figure 3-6a. The three benchmarks are: *histo_serial, Mandelbrot, and ann_training*. Figure 3-7 shows performance of optimized version relative to naïve version. Bars more than one show speed-up in the optimized version. The optimized benchmarks demonstrate performance improvement from 7% to 13%.

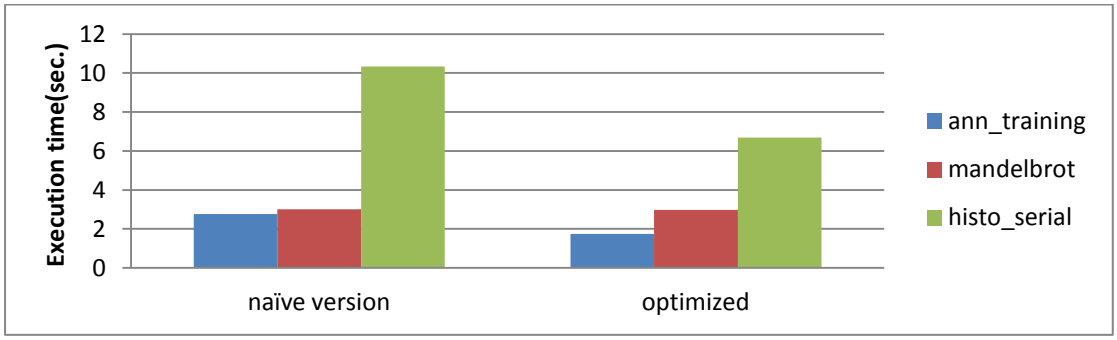

Figure 3-7: Performance of optimized and naively parallelized benchmarks.

3.2 Sensitivity of STM Programs to Static Parameters

NAS benchmark suite is a comprehensive set of benchmarks and covers a wide range of features of real world applications (Section 2.5). NAS benchmarks are designed using OpenMP library [39]. To convert NAS benchmarks into STM benchmarks, we replace critical sections in NAS with transactions. Figure 3-8 shows performance of STM version of NAS relative to the sequential version. Bars more than one show speed-up in STM version. Only two benchmarks are faster than the sequential version and others are all slower. On average, STM reduces performance by 43.8%, 57.5%, and 59.1%, when the number of threads is 2, 4, and 8 respectively.

We investigated the cause of slow-down in these benchmarks. In BT, MG, and FT, there are many large transactions which abort frequently. On the other side, IS is dominated by small transactions. The downside of small transactions is that overhead of STM APIs exceeds performance gain due to parallelism.

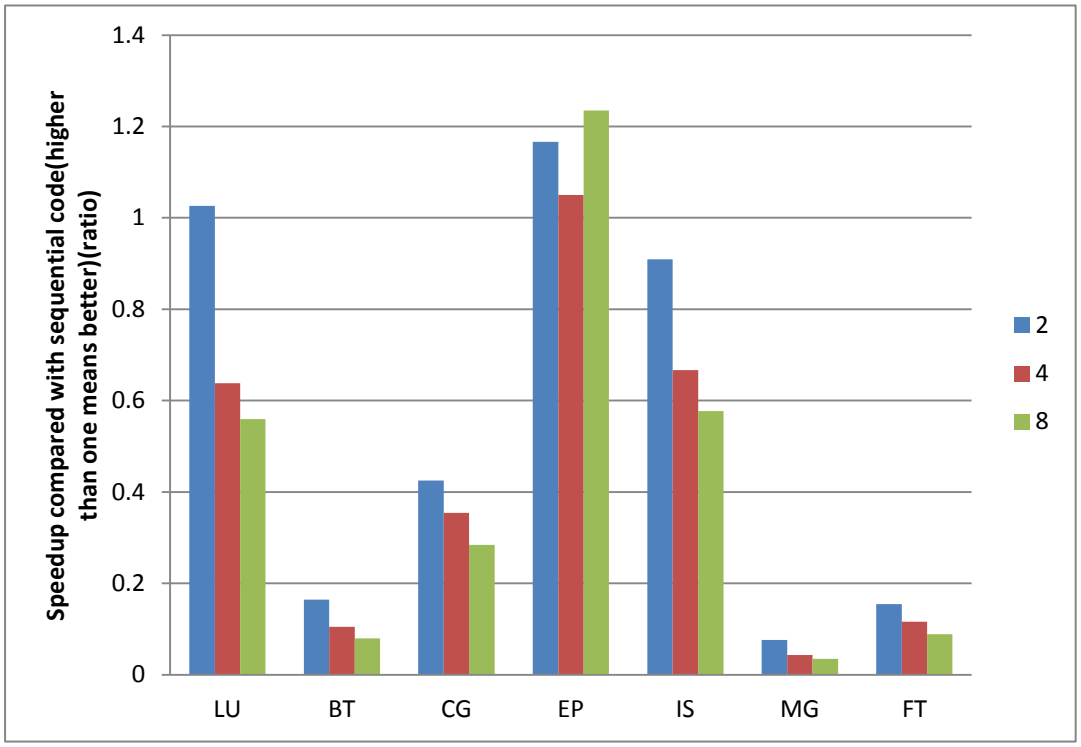

Figure 3-8: Speed-up in STM relative to sequential version of NAS benchmarks. The number of threads varies from two to 8.

In the next section, we focus on transaction size and explain how changing transaction size can improve performance of NAS benchmarks.

3.2.1 Transaction Size

Performance of STM programs varies with transaction size. Speed-up in a short transaction is limited since overhead of STM is high relative to the size of the transaction. On the other side, a long transaction may increase abort rate as a large number of instructions in a transaction may increase the window during which transactions are identified as competitors. So, to boost performance of STM programs, we should merge small transactions to reduce overhead of APIs. On the other side, we should split a large transaction into a number of small transactions to reduce abort rate and improve performance.

One way to measure transaction size is to count the number of C code lines in transactions. However, execution time of C programs changes from one line to the other by a large margin. We need a fine granularity metric for transaction size. Since

all C codes are compiled to assembly instructions, we use number of assembly instructions to measure transaction size.

To evaluate the impact of transaction size on performance, we use EigenBench [23] (Section 2.2). This micro-benchmark enables us to explore design space of STMs by orthogonally changing different transactional parameters. Unless otherwise specified, we use the parameters tabulated in Table 3-3 in our experiments.

Table 3-3: EigenBench parameters.

parameter	value	parameter	value
Thread number	2-8	Predominance	1.00
Temporal locality	0	Pollution	0.1
Working set size	256KB/thread	Density	1.0

In EigenBench, we can control the number of instructions per transaction through loop iterations. The micro-benchmark has a loop and the body of the loop is composed of three small loops. The number of iterations of the outer loop varies from 10 to 600. To measure the number of assembly instructions, we used gcc 4.8.1 to disassemble the C code.

Figure 3-9 shows performance of EigenBench when the number of instructions per transaction changes. We compare performance of STM version with sequential version. The number of threads varies from two to eight. We averaged execution time of the micro-benchmark over 10 runs.

Depending on the number of threads, the optimum transaction size changes. With 8 threads, the speed-up increases at the beginning rapidly, reaches to a maximum for 6784 assembly instructions, and then starts decreasing. When the number of threads is 4, the trend is totally different. The speed-up increases slightly at the beginning but steadily decrease when the number of assembly instructions exceeds 1696. The trend for 2 threads is almost the same as 4 threads. However, the highest speed-up for 2 threads is less and is around 1.

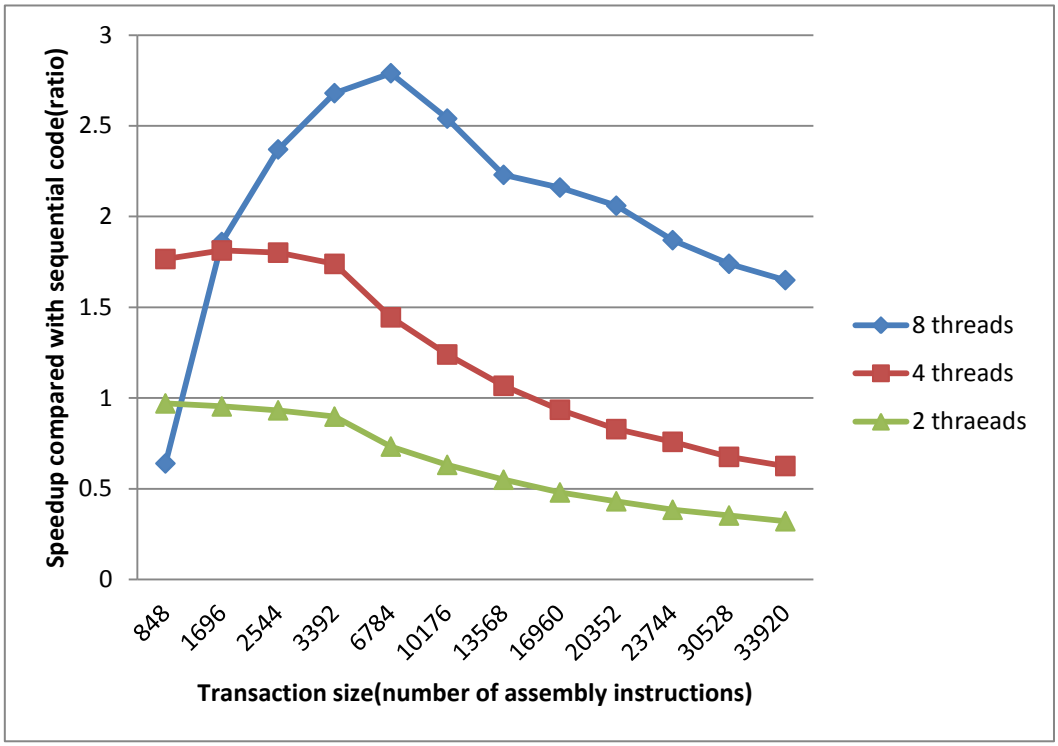

Figure 3-9: Speed-up when transaction size changes. The number of threads varies from two to eight.

Based on Figure 3-9, we can optimize STM programs. If a transaction is larger than the optimum size, then it should be broken into smaller transactions. On the other side, if a transaction is smaller than the optimum size, then it should be merged into other transactions. Figure 3-10 shows how to change size of a transaction. This is a code snippet taken from BT benchmark which is in NAS benchmark suite. Size of transaction is the number of assembly instructions between TM_BEGIN() and TM_END(). Therefore, to change transaction size, we can move TM_BEGIN() and TM_END() into appropriate locations. It is important to note that to guarantee correctness of the program, TM_SHARED_READ() and TM_SHARED_WRITE() must be between TM_BEGIN() and TM_END().

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

TM_END();
```

Figure 3-10: A code snippet taken from BT benchmark.

Based on Figure 3-9, the optimum transaction size when the number of threads is eight is 6784. However, in real applications, it is not always possible to change STM codes so that all transactions are optimum. For example, in Figure 3-10, transaction size is 5576. Since there is no other transaction in the code to merge with, the transaction cannot be changed to an optimum transaction.

We use the following two rules to optimize transaction size:

1. If a transaction is only 10% smaller or larger than 6784 assembly instructions, then we do not change the transaction.
2. We try to keep size of transactions between 3392 and 10176. If it is not possible, we refer to Figure 3-9 to determine the optimum size.

Figure 3-11 reports speed-up in optimized version of NAS benchmarks. Optimizing transaction size has dramatic impact on performance. For example, in BT, performance increases up to 9.3X. On average, changing transaction size improves performance by 77.7%, 88.4%, and 89.1% when the number of threads is 2, 4, and 8 respectively.

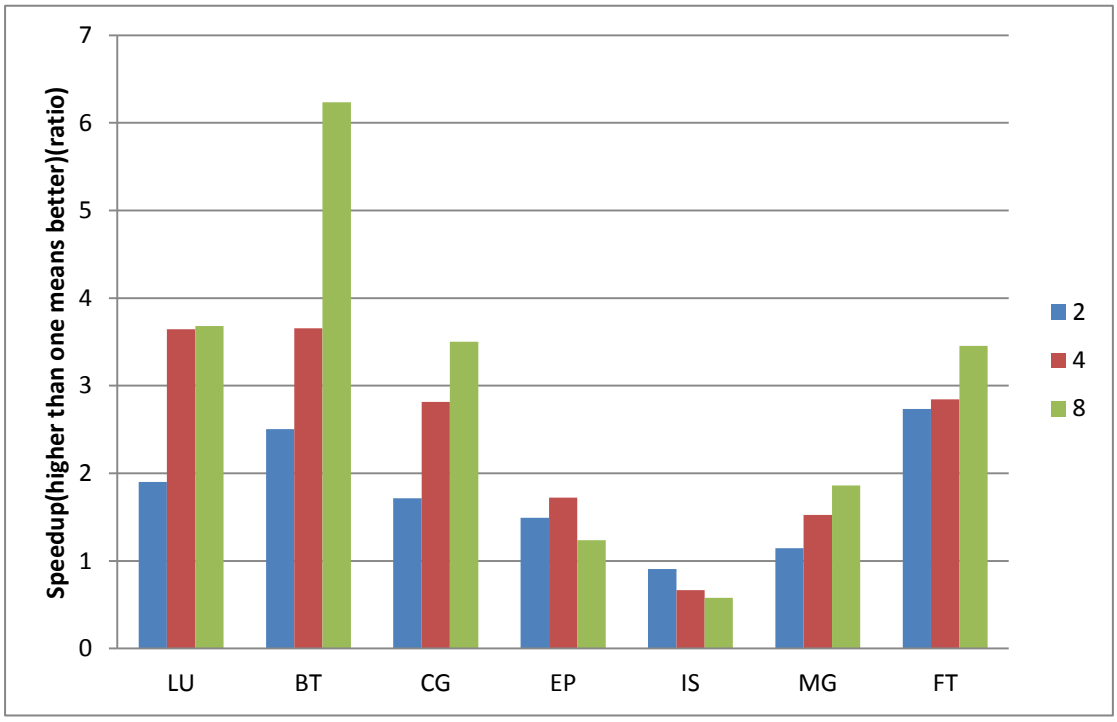

Figure 3-11: Speed-up in NAS benchmark suite when size of transaction is optimized.

There is an interesting phenomenon that IS still shows slowdown compared with sequential code. This is because of small transactions in IS. In IS benchmark, all the transactions are smaller than the optimum size. Furthermore, there are many non-transactional instructions between two consecutive transactions. Therefore, it is not possible to combine small transactions in this benchmark. So, in IS, the overhead of transactions (validation of read-set, lock acquisition, etc.) exceeds performance gain of parallelism and results in slow-down in this benchmark.

3.2.2 Size of Write-Set

TL2 uses write-set to record transactional write operations. The write-set is implemented through linked-list. When a transaction writes into a shared memory location, it inserts a new node to the linked-list. Each variable in the write-set is associated with a lock bit. In commit, the transaction traverses the linked-list to acquire locks and update memory with new transactional data.

Write-set is overhead of STMs as it does not exist in sequential programs. If a transaction fails to acquire a lock, then it aborts and restarts. So, a transaction with a

large write-set is more likely to abort. However, if we restrict transactions to have only small write-sets, then we need to split transactions into too many short transactions. This increases overhead of STM APIs relative to the performance gains of concurrent transactions and limits speed-up.

Similar to transaction size, we use EigenBench to evaluate the impact of write-set size on performance. EigenBench is not directly designed for evaluation of write-set size. The only parameter in EigenBench which affects write-set is pollution. Pollution is defined as the fraction of transactional writes. EigenBench keeps a fixed size transaction and increases or decreases the size of write-set to change the fraction of transactional writes. This is not what we require for evaluation of write-set size. Write-set is a linked-list which stores shared variables. So, it is not feasible to adjust size of write-set in a workload without changing the total number of memory locations accessed in the workload. If we increase or decrease the size of write-set through pollution, we may change the data structure of the workload.

The only way to optimize write-set is to split the transactions. Therefore, we created a new function based on pollution in EigenBench. The body of the function has a transaction with a large write-set. To change write-set size, we split the transaction into a number of small transactions. By measuring execution time of the function, we can evaluate the impact of write-set size on performance. Figure 3-12 shows the results.

Speed-up is lower for small and large write-sets. For small write-sets, the overhead of STM is higher than performance gain. In large data sets, there are more shared variables to check at commit stage. Therefore, the probability of conflicts increases.

In Figure 3-12, speed-up increases with the number of threads. This is mainly due to contention. A program with large number of threads has more conflicts, leading to higher probability to abort. This increases the impact of write-set optimization on performance.

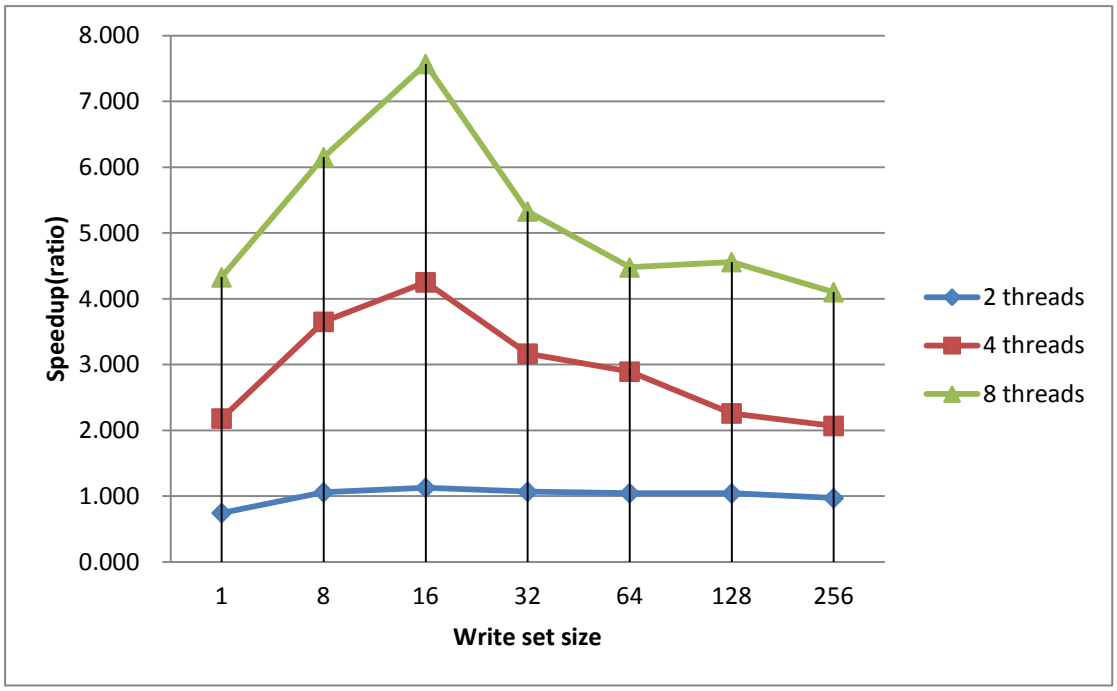

Figure 3-12: Performance of parallel EigenBench when write-set size changes.

Based on Figure 3-12, we optimized write-set of NAS benchmarks (Figure 3-13). Although the speed-up is not as high as optimized transaction size, it is still considerable. Optimizing write-set enhances performance up to 4.7X. The only benchmark which shows slowdown is IS. As we mentioned before, IS benchmark is composed of small transactions with many non-transactional instructions between them. This makes it impossible to create large transactions out of small ones.

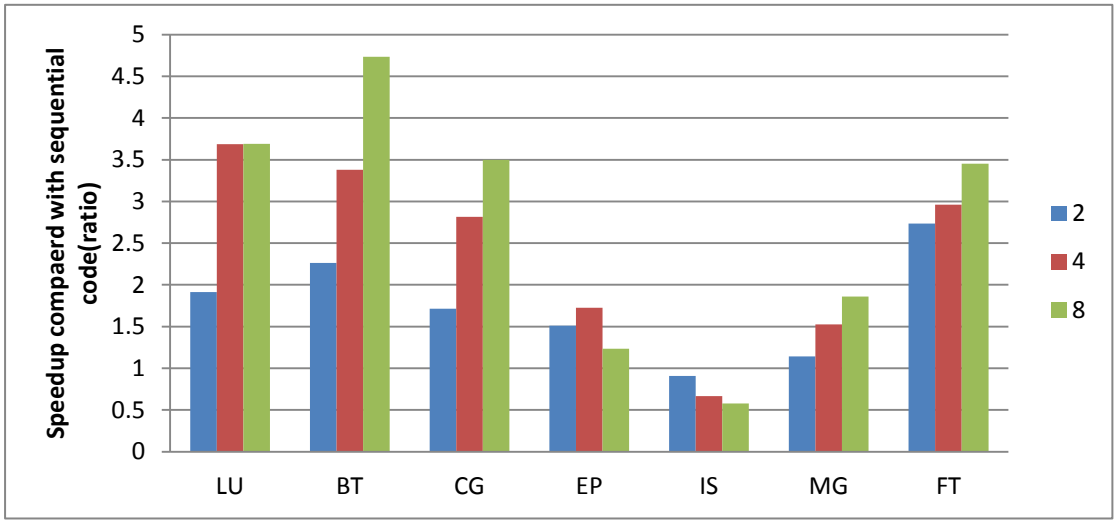

Figure 3-13: Speed-up in NAS benchmarks when write-set is optimized.

3.2.3 Size of Read-Set

Read-set in TL2 is used to store shared variables that are read by transactions. Since shared variables that are read by a transaction might be written by others, it is necessary to validate read-set in commit to guarantee atomicity of transactions. If validation fails then the transaction needs to abort and retry. Since a long read-set has more variables to validate than a short read-set, it is more likely that a long read-set results in abort. On the other side, while a short read-set reduces abort rate, it may increase overhead of STMs. Quite often, transaction size and read-set size are correlated. A small transaction accesses a few number of shared memory locations which results in small read-sets.

To evaluate the impact of read-set on performance, we use a function similar to the one that we used for write-set. The function is composed of several transactions. One of them is a read-only transaction and the rest are writing transactions. The initial read-only transaction has a large read-set. The transaction is broken into small transactions to create small read-sets. Other transactions are used to create contentions. Figure 3-14 shows speed-up of STM version of the function relative to the sequential version when read-set size changes.

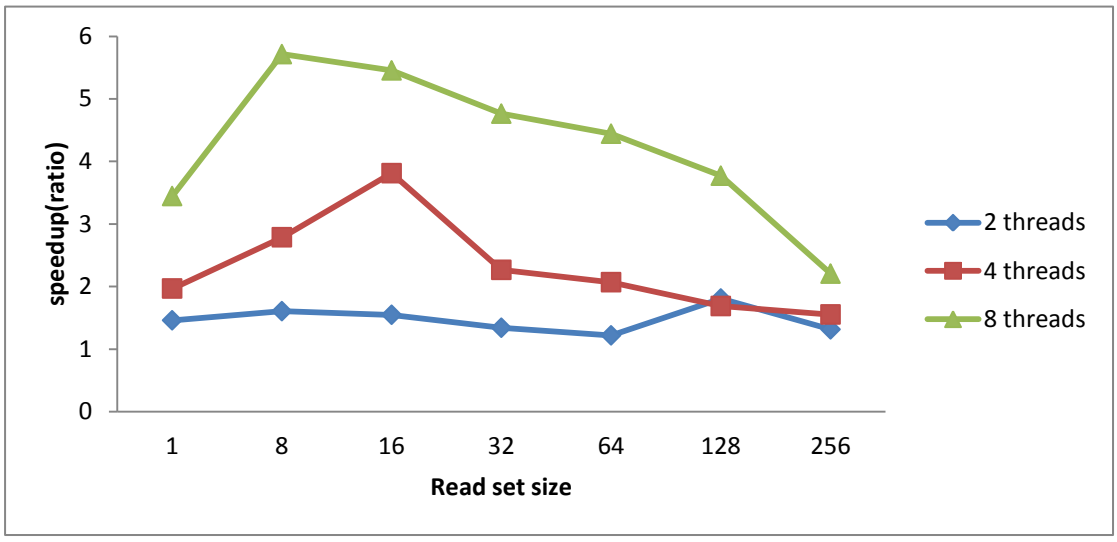

Figure 3-14: Speed-up in EigenBench where read-set size changes for 2, 4 and 8 threads.

Read-set size has significant impact on speed-up when the number of threads is 8. With 8 threads, speed-up is highest when the read-set size is 8. However, when the

number of threads is 2, the fluctuation is mild. As the number of threads reduces, there are fewer contentions among the running transactions. Hence, speed-up will be less sensitive to the read-set size. Another interesting point is the peak speed-up in each line. The peak with 8 threads is located at 8. The peak with 4 threads occurs when read-set size is 16. For 2 threads, the peak moves to read-set size of 128. So, as the number of threads reduces the peak point increases. This is mainly due to trade off between overhead of abort and overhead of STM APIs. When the number of threads is 8, the contention is high, so the overhead of abort can easily surpass the overhead of STM APIs. On the other side, when the number of threads is 2, the contention is low leading to lower abort overhead.

Based on Figure 3-14, we optimized NAS benchmark suite. However, there are a few read-only transactions in NAS benchmark suite. If we optimize both read-only and writing transactions, it is hard to specify whether speed-up is due to read-set or write-set. Therefore, there are few opportunities to optimize NAS benchmarks based on read-set. Figure 3-15 shows speed-up after optimizing the read-sets.

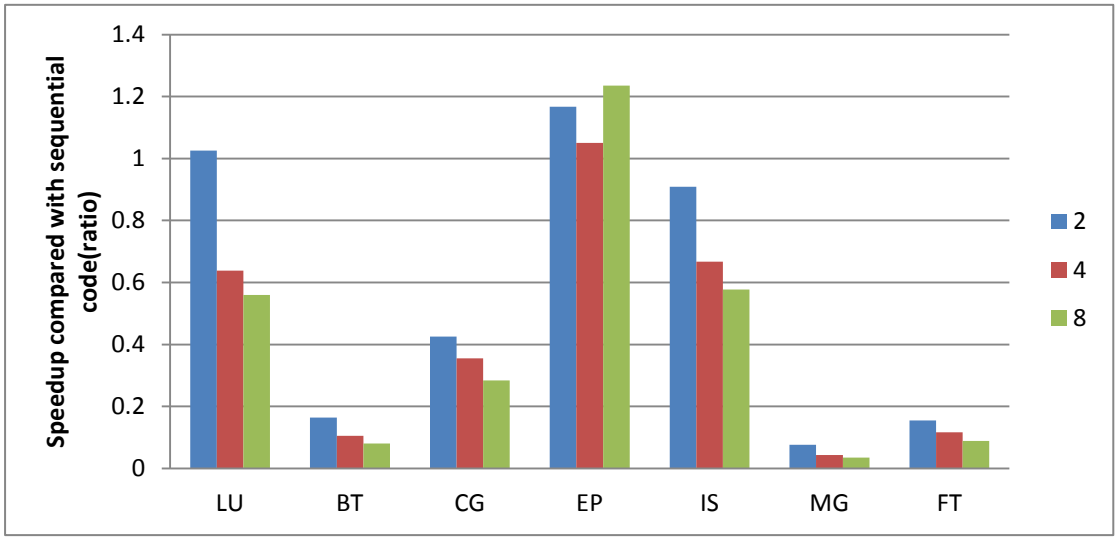

Figure 3-15: Speed-up in NAS benchmarks when read-set is optimized.

Based on Figure 3-15, read-set optimization slightly improves performance but it does not mean that read-set is less important than the other two factors. The main reason for small improvement is the limited number of read-only transactions in NAS

benchmarks. However, if we have a program which is dominated by read-only transaction, then optimizing read-sets result in higher speed-up.

3.2.4 Comprehensive Optimization Based on the Three Parameters

In this section we explain how these three parameters are optimized together. These parameters can impact each other. For example, a transaction with optimum transaction size may contain an oversized read-set or a small transaction may have a large write-set.

We optimized the three parameters one at a time. We start with transaction size and use Figure 3-9 to optimize transaction size. Then, we adjusted write-set size. To optimize write-set, we follow the rules below:

- If the write-set is shorter/larger than the optimum size, we enlarge/shrink the size of transaction step by step to approach the optimum size of the write-set. During this process, we measured execution time at each step. We select the transaction with minimum execution time and consider the transaction optimized for both transaction size and write-set size.
- If due to constraint in a benchmark, it is not feasible to optimize the write-set, we try to optimize both transaction size and write-set size simultaneously to find the minimum transaction size. For example, if a transaction contains a loop and the body of the loop is larger than optimum transaction size, we optimize transaction size and write-set size simultaneously.

In the next step, we optimized read-set. The rules for optimizing read-only transactions are similar to the above rules. However, for read-write transactions we follow the rules below:

- If a shared variable appears in both read-set and write-set, we cannot optimize read-set and write-set separately. First, we focus on read-set and size of transaction. Then, we change write-set size step by step and select the write-set that minimizes execution time.

- If read-set and write-set use disjoint variables, we can adjust size of read-set and size of write-set, separately. For read-set optimization, we enlarge or shrink the size of transaction gradually to reach the optimum read-set size. For write-set optimization, we follow rule 1 for write-set.

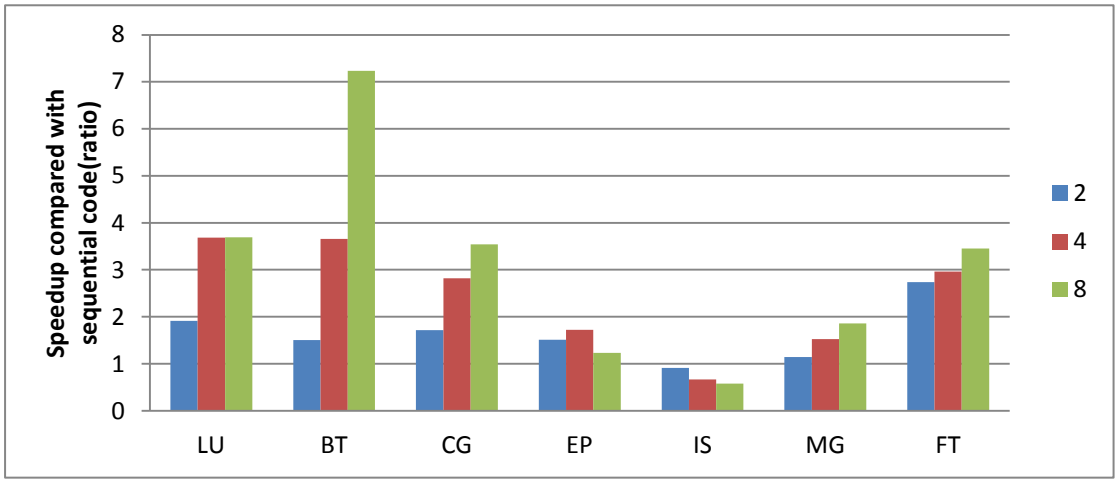

Figure 3-16: Speed-up for NAS benchmarks where size of transaction, write-set, and read-sets are optimized.

Figure 3-16 shows speed-up when all the three parameters are optimized. Based on Figure 3-16, optimization based on the three factors is better than optimization based on a single factor. Compared with non-optimized version, performance improves by 77.7%, 88.4%, and 89.1% when the number of threads is 2, 4, and 8, respectively.

3.3 Linear Regression Model

Optimizing an STM program based on the three parameters manually is a time consuming process. We need an automatic technique which can predict the optimum parameters for transactions. To optimize performance of STM programs, we build a linear regression model that predicts transaction size based on the three characteristics of a transaction: transaction size, write-set size, and read-set size. The main reason that we decided to use transaction size as the predicted value is that changing STM programs based on transaction size is straightforward. Quite often, it does not require any changes in the data structure of programs. For example, Figure 3-10 shows a code snippet from BT benchmark. The loop iterations are independent and so we can

change transaction size by splitting the outer loop into a number of smaller loops and assigning each small loop to a transaction. On the other side, changing write-set and/or read-set of a transaction needs significant programming effort which complicates parallel programming. Hence, in all our experiments, we target transaction size for optimization. It is important to note that in some programs, it is not feasible to break down a large transaction because of dependency. For example, if the loop iterations in Figure 3-10 are dependent, then we cannot break the outer loop.

Before deciding on using LR for prediction, we scrutinized three other techniques: SVM, decision tree, and neural networks. SVM and decision tree are not appropriate for prediction of transaction size as these techniques are only able to predict discrete variables. However, transaction size is a continuous variable. We also checked neural network for prediction. Theory and structure of neural network is more complex than linear regression. In addition, there is no common rule to design an optimum neural network. Neural network can have different number of layers and different propagation functions. On the other side, linear regression is simple to use and there are many matured applications that can be used to build linear regression model automatically. If we can get acceptable results from linear regression, then we can remove neural network from the list of candidates for prediction.

In this thesis, we use SPSS [40] to generate linear regression model. SPSS is a comprehensive and easy to use tool and helps users to optimize weight of each independent variable to find the best accuracy rate.

To simplify our discussion, we use 8 threads to evaluate our optimization techniques. In Section 3.6, we report experimental results for other number of threads.

3.3.1 Naive Version of Linear Regression Model

The first version of LR model uses three factors as input variables: non-optimized transaction size, non-optimized write-set size, and non-optimized read-set size. The predicted output is optimum transaction size. We selected 40 transactions from NAS

benchmark suite for training. For test, we use DP benchmark suite [12] and Stamp benchmark suite [18].

After training, SPSS generated equation 3-1 for prediction. Tx_p stands for predicted transaction size. TxSize, WrSize and RdSize are the parameters of non-optimized transactions. We selected 6 transactions from DP benchmark suite to test the LR model. Table 3-4 shows predicted transaction size. On average, error of prediction is 45.32%. In some benchmarks, the accuracy error reaches to -335.26%.

$$Tx_p = 7513.761 + 0.07TxSize - 5WrSize - RdSize \quad (3\text{-}1)$$

Table 3-4: Accuracy of Predictions in Naive LR Model.

TX	No optimized TX size	Predicted TX Size	Optimum TX Size	Error (%)
Test 1	148258	16859.82	6739	-150.18%
Test 2	54736	10829.28	2488	-335.26%
Test 3	112816	15122.88	5128	-194.91%
Test 4	636460	51735.96	28930	-78.83%
Test 5	204192	21675.2	6381	-239.68%
Test 6	35122	9846.301	35122	71.97%

The LR model which only relies on transaction, write-set, and read-set sizes results in low accuracy rate. We analyzed structure of STM programs and found that optimization of current transaction is closely related to the next transaction. For example, if we try to combine small transactions, we need to consider the distance between two transactions. If there are only a few sequential instructions between the two transactions, we can combine them by including the sequential instructions in the final transaction. However, if there are a large number of instructions between the two transactions, we need to consider the impact of those sequential instructions on transaction size. The other example is when a transaction needs to enlarge but the next transaction is optimized. In such a case, the non-optimized transaction cannot be changed. So, we needed to consider these factors in LR prediction and revise our model.

After several experiments, we decided to extend the inputs of the LR and include five more parameters: size of next transaction (SNT), number of assembly instructions

between two consecutive transactions (NCT), write-set of the next transaction (WN), read-set of the next transaction (RN), and number of assembly instructions in a loop (TL). These five parameters are in addition to the original three parameters: size of transaction (ST), size of write-set (WS), and size of read-set (RS)

The first factor is called SNT. We explain why we use SNT as an input to the LR model through an example. Assume that transaction A is followed by transaction B and transaction C is followed by transaction D. Transactions A, B, C, and D have 3000, 5000, 6000, and 11000 instructions, respectively, Assume that the optimum transaction size is 8000 instructions. We can combine transactions A and B and create a larger transaction with 8000 instructions. However, transactions C and D cannot be combined since the combined transaction has much more than 8000 instructions.

The second factor is called NCT. The number of instructions between two consecutive transactions affects the way we merge multiple small transactions into a large transaction. Assume that there are two transactions each with 3000 instructions. Similar to the previous example, assume that the optimum transaction size is 8000 instructions. If NCT is 2000 instructions, then the combined transaction results in optimum performance. However, if NCT is 10000, then we cannot combine the two transactions as the combined transaction is too large and hurts performance.

The third and fourth parameters are called WN and RN. Similar to SNT, write-set and read-set of the next transaction affect how we merge small transactions to build optimum transactions. So, to optimize transaction size, we need to consider WN and RN as well.

The fifth parameter is called TL. This parameter affects those transactions that are inside the body of a loop. If the total number of instructions in a loop is less than optimum transaction size, then we can move the whole loop into a transaction. For transactions that are not inside a loop, we set this parameter to zero.

Equation 3-2 shows LR model using the 8 input parameters. We used SPSS [40] to calculate coefficients in equation 3-2. TS stands for transactions size.

$$TS = 5451 + 0.867 \times \mathrm{ST} + 0.015 \times \mathrm{RS} - 0.027 \times \mathrm{WS} - 0.832 \times \mathrm{TL} + 0.229 \times \mathrm{NCT} + 0.032 \times \mathrm{SNT} + 0.015 \times \mathrm{RN} - 0.026 \times \mathrm{WN} \quad (3\text{-}2)$$

Table 3-5 shows accuracy of predictions by LR. The test cases in Table 3-5 are transactions from DP benchmarks which are the same as those test cases used for evaluation of the naive version. While accuracy is high in some of the benchmarks, i.e. test 6, in most of the benchmarks, LR prediction still results in significant error. The main reason for high error is that LR tries to draw a line to cover as many points as possible. If the points are scattered, then LR is unable to fit a line that covers all the points. This reduces accuracy of predictions.

Table 3-5: Accuracy of predictions in the revised LR model.

TX	No Optimized TX Size	Predicted TX Size	Optimum TX Size	Error (%)
Test 1	148258	10576.54	6739	-56.90%
Test 2	54736	6985	2488	-180.70%
Test 3	112816	9159	5128	-78.60%
Test 4	636460	27062	28930	6.50%
Test 5	204192	5343	6381	16.30%
Test 6	35122	34385	35122	2.10%

3.3.2 Multi-linear Regression Model

Further investigation of LR model reveals that the error rate for transactions with large negative error is in the range of 56%-180.7%. On the other side, error rate of transactions with large positive error is in the range of 6.5%-16.3%. This motivates us to classify transactions into three categories: transactions with large negative error (class1), transactions with large positive error (class2), and transactions with small error (clas3). We use separate LR model for each class. This improves accuracy of predictions since the set of points within a class are well-organized and fitting a curve to the points results in less residual error. We use the same 8 input parameters for the three LR models: SNT, NCT, WN, RN, TL, ST, WS, and RS. Equations 3-3 to 3-5

show the new LR models. TS1, TS2, and TS3 correspond to predicted TX size in class1, class2, and class3, respectively.

$$TS_1 = 416 + 0.013 \times RS - 0.020 \times \mathrm{WS} - 0.043 \times \mathrm{TL} + 97.09 \times \mathrm{NCT} + 0.041 \times \mathrm{SNT} + 0.012 \times \mathrm{RN} - 0.019 \times \mathrm{WN} \quad (3\text{-}3)$$

$$TS_2 = 7196 + 0.824 \times ST + 0.018 \times RS - 0.033 \times WS - 0.791 \times TL - 0.015 \times NCT + 0.03 \times SNT + 0.018 \times RN - 0.031 \times WN \quad (3\text{-}4)$$

$$TS_3 = 8142 + 0.799 \times ST + 0.024 \times RS - 0.039 \times WS - 0.765 \times TL - 0.026 \times NCT + 0.03 \times SNT + 0.023 \times RN - 0.035 \times WN \quad (3\text{-}5)$$

To evaluate the accuracy rate of this multi-LR model, we used the same six test cases of single LR model. Table 3-6 shows predictions made by multi-LR model. Under multi-LR model, accuracy of predictions increases significantly. On average, error rate drops from 59% to 1.7%. The maximum and minimum error rates are 7.22% and 0%, respectively. In four out of six test samples, error rate is less than 0.5%.

Table 3-6: Accuracy of Predictions Made by multi-LR model.

name	None Optimized TX Size	Predicted TX Size	Optimum TX Size	Error (%)
Test 1	148258	6739	6739	0%
Test 2	54736	2488.45	2488	-0.02%
Test 3	112816	5129.87	5128	-0.04%
Test 4	636460	28930	28930	0%
Test 5	204192	5919.92	6381	7.22%
Test 6	35122	34247	35122	2.49%

3.4 Classifier for multi-LR model

We need a classifier to decide which LR model should be used for a transaction. For the above evaluations, we selected the LR models manually. There are only 6 test cases and so manual process is not time consuming. However, for a large number of test cases, manually selecting the LR models is not feasible. We need a method that

automatically selects the appropriate LR model based on characteristics of transactions.

In Section 2.4, we discussed several classifiers. Two popular classifiers are decision tree and SVM. There are also enhanced decision tree and SVM using boost techniques such as adaboost [34].We can combine adaboost with decision tree or SVM to improve accuracy of predictions. In the rest of this section, we compare the accuracy of three different classification models: decision tree, SVM, and decision trees boosted with adaboost.

3.4.1 Decision Tree

To classify transactions based on decision tree, we use C4.5 [26]. C4.5 is a popular decision tree algorithm which is able to classify objects with continuous attributes. We train the decision tree with already classified sample transactions. Each sample S_i consists of an 8-dimensional input vector (SNT, NCT, WN, RN, TL, ST, WS, and RS) as well as the class which S_i belongs to. Through the training phase, the decision tree learns how to classify transactions. For test, we feed the decision tree an 8-dimensional vector and the decision tree predicts the class of the transaction corresponding to the vector.

Figure 3-17 shows the output of C4.5 when the number of threads is 8. We will report output of C4.5 for other number of threads in section 3.6. According to Figure 3-17, LR model 1 is selected when transaction size is less than or equal to 3074 and the read-set size is less than or equal to 7. By following the output of C4.5, we can classify transactions of an STM program.

To evaluate C4.5, we use the 6 test cases which are used in the previous evaluations. Table 3-7 illustrates predictions for the 6 test cases. C4.5 mispredicts only one test case. For all other test cases, C4.5 accurately predicts the LR model.

```
Decision tree:
INPUT: SNT, NCT, WN, RN, TL, ST, WS, and RS

IF transaction size <= 3074
    THEN IF read-set size <= 7
        THEN select regression model 1
        ELSE select regression model 3
ELSE
    THEN IF read-set size <= 64
        THEN select regression model 2
    ELSE
        THEN IF transaction size <= 200673
            THEN select regression model 3
            ELSE select regression model 2
```

Figure 3-17: Output of C4.5 for multi-LR model.

Table 3-7: Classification based on decision tree.

TX	Correct Model	Predicted Model
Test 1	Model 3	Model 3
Test 2	Model 2	Model 2
Test 3	Model 3	Model 3
Test 4	Model 3	Model 2
Test 5	Model 2	Model 2
Test 6	Model 2	Model 2

3.4.2 SVM Classifier

To classify transactions based on SVM, we used libsvm V.3.2 [41]. Since some transactions have large transactions with small write-sets or read-sets, scale problem can affect accuracy of predictions. To minimize the scale problem, we call normalization function [41] before training the SVM with datasets.

Table 3-8: Classification based on SVM.

TX	Correct Model	Predicted Model
Test 1	Model 3	Model 2
Test 2	Model 2	Model 3
Test 3	Model 3	Model 2
Test 4	Model 3	Model 3
Test 5	Model 2	Model 2
Test 6	Model 2	Model 2

Table 3-8 shows prediction results for the six test cases using libsvm V.3.2. The accuracy of SVM is lower than decision tree. This is because SVM is sensitive to noisy dataset. In training dataset, usually, most of small transactions belong to LR model 1. However, a few small transactions belong to LR model 3. This creates noise and confuses the SVM. Since decision tree is more resilient to noise than SVM, its accuracy is higher.

3.4.3 Adaboost Decision Tree

We used C4.5 as classifier in adaboost technique. Adaboost is implemented in MATLAB 2014b. Table 3-9 shows accuracy of predictions in adaboost. On average, the accuracy rate is 66.7%. Similar to SVM, adaboost falls behind decision tree because of nosy dataset. However, accuracy of adaboost is higher than SVM.

Table 3-9: Classification based on adaboost.

TX	Correct Model	Predicted Model
Test 1	Model 3	Model 3
Test 2	Model 2	Model 2
Test 3	Model 3	Model 2
Test 4	Model 3	Model 2
Test 5	Model 2	Model 2
Test 6	Model 2	Model 2

3.5 Mixed Decision Tree and Multi-Linear Regressions Model

To predict transaction size, we use a combination of linear regression and decision tree. First, decision tree determine the class of a transaction. Then, we use one of the three LR models (equations 3-3 to 3-5) to predict optimal transaction size.

Figure 3-18 shows the steps that should be taken to find out the optimum transaction size. First, the input vector corresponding to a transaction is determined through profiling. Then, a classifier determines which LR model should be used for prediction. In the next step, the selected LR model predicts the optimum transaction size.

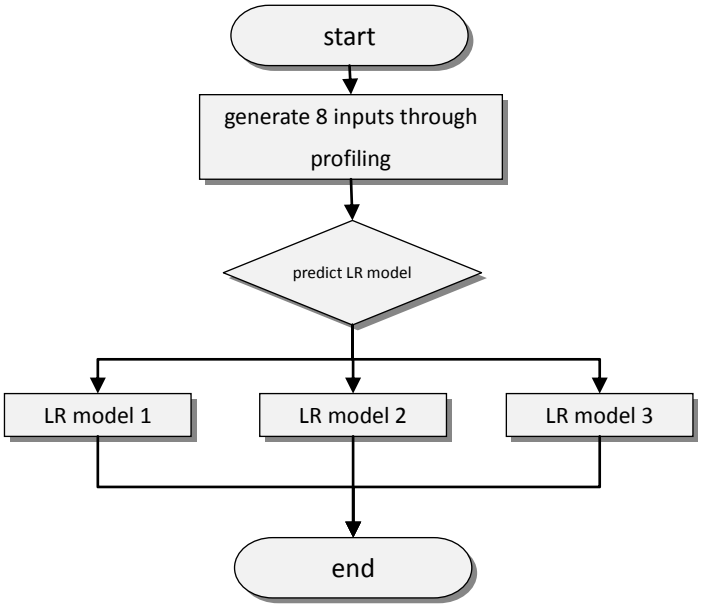

Figure 3-18: Flow chart for predicting transaction size.

3.6 Details of Mixed Models for Other Number of Threads

So far, we focused on 8 threads in all our evaluations. The optimum transaction size, write-set size, and read-set size depend on number of threads. So, LR models and also classifier change with the number of threads. In this section, we report experimental results for two and four threads.

3.6.1 Mixed Model for 2 Threads

To build LR models, we use the same 8 parameters for prediction. Optimum transaction size for 2 threads is greater than 8 threads. So, we use four LR models instead of three for 2 threads. Two out of four are the same: an LR model for large negative error and an LR model for small error. The other two LR models are used for large and medium positive errors. Equations 3-6, 3-7, 3-8, and 3-9 correspond to, small, medium, large positive and large negative errors.

$$TS_1 = 8561.396 + 1.174 \times ST - 1.242 \times RS - 1.223 \times WS - 0.945 \times TL + 0.022 \times NCT - 0.023 \times SNT - 0.349 \times RN + 0.337 \times WN \quad (3\text{-}6)$$

$$TS_2 = 27985.805 - 0.027 \times ST - 1.517 \times \mathrm{WS} + 0.169 \times \mathrm{TL} + 0.504\mathrm{SNT} + 0.056 \times \mathrm{RN} + 0.299 \times \mathrm{WN} \quad (3\text{-}7)$$

$$TS_3 = 57718.722 + 0.426 \times ST - 3.705 \times RS - 3.291 \times WS - 0.136 \times SNT - 3.496 \times RN + 5.632 \times WN \quad (3\text{-}8)$$

$$TS_4 = 343.453 + 0.312 \times ST + 0.262 \times RS + 0.03 \times \mathrm{WS} + 0.428 \times \mathrm{TL} - 0.034 \times \mathrm{NCT} + 0.138\mathrm{SNT} + 0.301 \times \mathrm{RN} + 0.032 \times \mathrm{WN} \quad (3\text{-}9)$$

Since we have 4 LR models for 2 threads, we need to rebuild the decision tree. Figure 3-19 shows the output of decision tree.

```
Decision tree:
INPUT: SNT, NCT, WN, RN, TL, ST, WS, and RS

IF transaction size <= 2488
    THEN select regression model 1
ELSE
    THEN IF write-set size >= 135
        THEN select regression model 2
    ELSE IF next write-set size >101
            THEN select regression model 4
        ELSE IF length to next transaction > 83
                THEN select regression model 3
            ELSE IF transaction size<=33695
                    THEN select regression model 4
                ELSE select regression model 2
```

Figure 3-19: Decision tree model for multi-LR model.

3.6.2 Evaluation of Mixed Model for 2 Threads

Tables 3-9, 3-10, and 3-11 show predictions by decision tree, SVM, and adaboost, respectively. For 2 threads, decision tree is slightly better than SVM and adaboost. Compared to 8 threads, accuracy rate of decision tree and SVM reduces. This is mainly due to increased number of LR models. Because we have 4 categories, the boundary of each category is not as clear as 8 threads. In other words, the influence of noise reduces accuracy rate.

Table 3-10: Predictions made by decision tree for 2 threads.

TX	Correct model	Predicted model
Test 1-2 threads	Model 2	Model 2
Test 2-2 threads	Model 3	Model 3
Test 3-2 threads	Model 4	Model 3
Test 4-2 threads	Model 3	Model 3
Test 5-2 threads	Model 4	Model 2
Test 6-2 threads	Model 2	Model 2

Table 3-11: Predictions made by SVM for 2 threads.

TX	Correct model	Predicted model
Test 1-2 threads	Model 2	Model 2
Test 2-2 threads	Model 3	Model 3
Test 3-2 threads	Model 4	Model 3
Test 4-2 threads	Model 3	Model 3
Test 5-2 threads	Model 4	Model 2
Test 6-2 threads	Model 2	Model 2

Table 3-12: Predictions made by adaboost for 2 threads.

TX	Correct model	Predicted model
Test 1-2 threads	Model 2	Model 3
Test 2-2 threads	Model 3	Model 4
Test 3-2 threads	Model 4	Model 4
Test 4-2 threads	Model 3	Model 3
Test 5-2 threads	Model 4	Model 2
Test 6-2 threads	Model 2	Model 2

3.6.3 Mixed Model for 4 Threads

In this section, we present details of mixed model for 4 threads. Unlike 2 threads, we can use 3 LR models to express the whole dataset for 4 threads. Equations 3-10 to equation 3-12 are the LR models for 4 threads corresponding to large positive, small, and large negative errors.

Figure 3-20 shows the output of C4.5 for 4 threads. Tables 3-12, 3-13, and 3-14 show predictions made by decision tree, SVM, and adaboost, respectively. Adaboost has the best accuracy rate but SVM still has the lowest accuracy rate. The Noise in 4 threads is less than 2 threads as we have only three categories for classification. Also, for 4 threads, the optimum transaction size is more than 8 threads. So, the boundary between each category is clearer for 4 threads. Low level of noise and clear boundary are suitable for adaboost. However, lowest accuracy rate of SVM indicates that this level of noise is still too high for SVM.

$$TS_1 = 17309.641 - 0.48 \times ST + 0.887 \times RS - 0.864 \times WS - 0.065 \times TL + 1.326 \times NCT + 0.041SNT + 1.165 \times RN + 1.133 \times WN \quad (3\text{-}10)$$

$$TS_2 = 16197.612 - 0.164 \times ST - 5.252 \times WS + 0.939 \times TL0.003 \times SNT - 0.349 \times RN + 0.230 \times WN \quad (3\text{-}11)$$

$$TS_3 = 24.345 + 1.174 \times ST + 2.753 \times RS - 1.242 \times WS - 0.443 \times TL + 0.022 \times NCT - 0.107 \times SNT - 0.501 \times RN + 0.143 \times WN \quad (3\text{-}12)$$

```
Decision tree:
INPUT: SNT, NCT, WN, RN, TL, ST, WS, and RS

IF total loop <= 6081
    THEN select regression model 3
ELSE
    THEN IF size of next transaction >= 61013
        THEN select regression model 1
    ELSE IF next write-set size <=16
            THEN select regression model 2
        ELSE IF size of transaction > 25760
                THEN select regression model 1
            ELSE IF total loop<=53655
                    THEN select regression model 2
                ELSE select regression model 1
```

Figure 3-20: Output of decision tree for 4 threads.

Table 3-13: Predictions made by decision tree for 4 threads.

TX	Correct model	Predicted model
Test 1-4 threads	Model 2	Model 1
Test 2-4 threads	Model 1	Model 1
Test 3-4 threads	Model 2	Model 2
Test 4-4 threads	Model 1	Model 1
Test 5-4 threads	Model 1	Model 1
Test 6-4 threads	Model 2	Model 2

Table 3-14: Predictions made by SVM for 4 threads.

TX	Correct model	Predicted model
Test 1-4 threads	Model 2	Model 2
Test 2-4 threads	Model 1	Model 1
Test 3-4 threads	Model 2	Model 1
Test 4-4 threads	Model 1	Model 1
Test 5-4 threads	Model 1	Model 2
Test 6-4 threads	Model 2	Model 2

Table 3-15: Predictions made by adaboost for 4 threads.

TX	Correct model	Predicted model
Test 1-4 threads	Model 2	Model 2
Test 2-4 threads	Model 1	Model 1
Test 3-4 threads	Model 2	Model 2
Test 4-4 threads	Model 1	Model 1
Test 5-4 threads	Model 1	Model 1
Test 6-4 threads	Model 2	Model 2

3.7 Summary of contributions

In this chapter, we evaluated the impact of transactional parameters on performance of STM programs. We proposed mixed models to predict optimal transaction size. As optimal transaction size depends on number of threads, we generated three mixed models for 2, 4 and 8 threads. In chapter 4, we evaluate accuracy of the mixed models and report speedup.

Chapter 4

Experimental Results

In this chapter, we report performance of the mixed prediction model. To measure the performance, we selected two indicators: accuracy rate of prediction and speed-up. Accuracy rate shows how often a model can classify different types of workloads properly. Speed-up is defined as execution time of the baseline scheme divided by execution time of the enhanced scheme and shows whether a model can boost execution time of applications.

4.1 Benchmark Suites

We used a subset of NAS [11] and DiscoPoP benchmark suites [12] to train LR and decision tree. To test our models, we used the rest of the benchmarks from NAS and DiscoPoP benchmark suites and also benchmarks from Stamp benchmark suite.

Stamp benchmark suite [18] is designed by Stanford University for shared memory parallel applications. This benchmark suite contains 8 benchmarks: *bayes, genome, intruder, kmeans, labyrinth, ssca2, vacation, and yada*. Here is a brief description of these benchmarks:

- Bayes: an algorithm to build a bayesian classification model.
- Genome: an application for gene sequencing.
- Intruder: used in domain of security which can monitor intrusions in computer networks.
- Kmeans: an algorithm for data mining,
- Labyrinth: an algorithm to find routes in a maze.
- Ssca2: this benchmark is used to generate efficient graph representation.
- Vacation: an algorithm to simulate travel reservation.

- Yada: this benchmark is used to refine Delaunay mesh.

4.2 Speed-up for DP Benchmarks

Tables 4-1, 4-2, and 4-3 show predicted transaction size for 2, 4, and 8 threads, respectively. If a benchmark has more than one transaction, then the name of the benchmark is followed by transaction number to distinguish different transactions.

Table 4-1: Predicted and Optimum TX Size in DP for 2 threads.

TX	None Optimized TX Size	Predicted TX Size	Optimum TX Size
Histo_serial	320625	15523	14800
Mc_light-Tx1	2125000	18120	5673
Mc_light-Tx2	465000	9076	10130
Ann_trainig-Tx1	288000	14157	11772
Ann_trainig-Tx2	480000000	748569	12725
Mandelbrot	78208	17158	16344

Table 4-2: Predicted and Optimum TX Size in DP for 4 threads.

TX	None Optimized TX Size	Predicted TX Size	Optimum TX Size
Histo_serial	320625	16267	9672
Mc_light-Tx1	2125000	52633	5673
Mc_light-Tx2	232500	14148	10130
Ann_trainig-Tx1	144000	9544	11772
Ann_trainig-Tx2	480000000	1225201	12725
Mandelbrot	78208	12255	16344

Table 4-3: Predicted and Optimum TX Size in DP for 8 threads.

TX	None Optimized TX Size	Predicted TX Size	Optimum TX Size
Histo_serial	320625	14186	7440
Mc_light-Tx1	2125000	77310	5673
Mc_light-Tx2	116250	11148	10130
Ann_trainig-Tx1	72000	10614	11772
Ann_trainig-Tx2	480000000	16328112	12725
Mandelbrot	78208	9776	8673

In some benchmark, the error of prediction is very large. For example, the error rate in ann_training-TX2 is 578% when the number of threads is two. This transaction is a large transaction. However, in our training dataset, we do not have such a large transaction. This reduces accuracy of prediction. Histo_serial is another benchmark that its error rate is high. This benchmark has a large transaction with small write-set. We also do not have samples similar to histo_serial in our training dataset.

To measure speed-up, we optimize all transactions in the benchmarks according to predictions. Each benchmark is run 10 times and the average of execution times is calculated. Figure 4-1 shows speed-up in DP benchmarks. Bars greater than one show speed-up in optimized code. On average, performance is improved by 43.75%, 59.50%, and 42.10% when the number of threads is 2, 4, and 8, respectively.

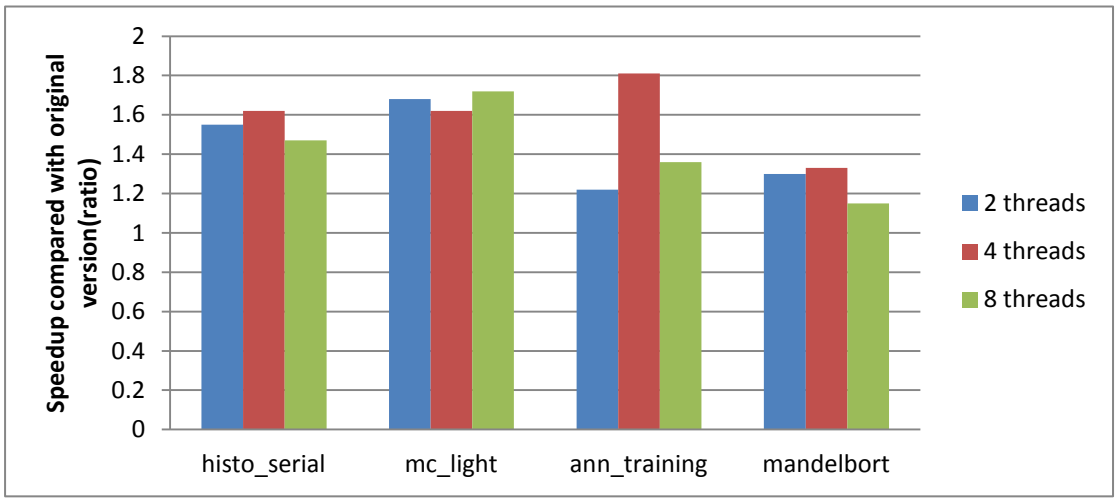

Figure 4-1: Speed-up for DP benchmarks.

4.3 Speed-up for Stamp Benchmarks

In Stamp benchmark suite, *genome, kmeans*, and *ssca2* cannot be optimized due to dependency among transactional variables. Only *bayes, vacation*, and *yada* can be optimized. Table 4-4 to 4-6 show the results of predicted transaction size and speed-up compared with the baseline scheme. DT means using decision tree as classifier and ADA means using adaboost decision tree as classifier.

Table 4-4: Transaction size and speed-up in Stamp benchmark suite for 2 threads.

TX	None Optimized TX Size	Predicted TX Size	Speed-up
bayes-DT	1261	16747	14.83%
vacation-DT	672	9925	9.14%
yada-DT	573	5452	11.41%
bayes-SVM	1261	16747	14.83%
vacation-SVM	672	9925	9.14%
yada-SVM	573	1125	0%
bayes-ADA	1261	16747	14.83%
vacation-ADA	672	847	0%
yada-ADA	573	5452	11.41%

Table 4-5: Transaction size and speed-up in Stamp benchmark suite for 4 threads.

TX	None Optimized TX Size	Predicted TX Size	Speed-up
bayes-DT	1261	12745	22.25%
vacation-DT	672	7975	6.65%
yada-DT	573	8183	18.32%
bayes-SVM	1261	12745	22.25%
vacation-SVM	672	7975	6.65%
yada-SVM	573	8183	18.32%
bayes-ADA	1261	12745	22.25%
vacation-ADA	672	7975	6.65%
yada-ADA	573	8183	18.32%

Table 4-6: Transaction size and speed-up in Stamp benchmark suite for 8 threads.

TX	None Optimized TX Size	Predicted TX Size	Speed-up
bayes-DT	1261	8218	39.60%
vacation-DT	672	7238	7.20%
yada-DT	573	7223	26.30%
bayes-SVM	1261	8218	39.60%
vacation-SVM	672	7238	7.20%
yada-SVM	573	7223	26.30%
bayes-ADA	1261	8218	39.60%
vacation-ADA	672	7238	7.20%
yada-ADA	573	456	0%

Across all benchmarks, decision tree always has better accuracy rate than adaboost. In some benchmarks, speed-up is 0%. In these benchmarks, due to restrictions in the structure of transactions, it is not feasible to optimize transaction size. On average, performance is improved by 9.51%, 15.74%, and 21.45%, when the number of threads is 2, 4, and 8, respectively.

Chapter 5

Conclusion and Future work

5.1 Conclusion

In this thesis, we presented an optimization technique that helps programmers to write efficient STM programs. We studied the impact of three parameters on STM performance and showed that STM applications are highly sensitive to the three parameters. Then, we exploited LR to predict transaction size based on the three parameters. A single LR model is not accurate enough and it results in high error rate. We revised the LR model by extending its inputs from 3 to 8 parameters. Also, to improve accuracy of LR, we classified transactions into three groups for 4 and 8 threads: transactions with large positive errors, transactions with large negative errors, and transaction with low errors. For 2 threads, there is an extra category: transactions with medium positive errors.

Our evaluations using DP and Stamp benchmark suites show that the mixed model is effective and is able to improve performance of transaction applications. On average, the mixed model improves performance of DP and Stamp benchmark suites by 48.45% and 15.56%, respectively.

6.2 Future Work

The mixed model is able to predict the optimum transaction size. However, a programmer needs to change transaction size manually. This is a time consuming process and requires significant programming effort. If the mixed model is integrated with a compiler, then the optimization process can be done automatically without interference of programmers. DiscoPoP (Section 2.2) is a compiler that can find parallel parts of a sequential code. One possibility for future work is changing DiscoPoP so that it automatically converts a sequential code to an optimized STM program. Given that most of software packages are written sequentially, an optimizing

compiler such as DiscoPoP provides an ample opportunity to use STM for commercial applications.

The other way to extend our work is using partial rollback. When a transaction aborts, all instructions executed in the transactional section are aborted. However, some of those instructions may generate the same output when they are executed again. Re-executing these instructions is wasteful and underutilizes precious processor resources. We can use a static approach and mark those instructions that do not need to re-execute. To integrate partial rollback with our LR model, we need to revise our model and add extra parameters for training. Furthermore, EigenBench does not support partial rollback. So, we need to change EigenBench to individually evaluate the impact of different parameters on performance in an STM system with rollback support.

References

[1] H. Maurice, Moss, J. Eliot B, "Transactional memory: Architectural support for lock-free data structures", in the *20th International Symposium on Computer Architecture (ISCA)* ,1993, .pp. 289–300

[2] N.Shavit, and T. Dan, "Software transactional memory." in *Distributed Computing,* 2 October 1997, pp. 99-116.

[3] P. Damron, A. Fedorova, Y. Lev, V. Luchangco,M. Moir, and D. Nussbaum, " Hybrid transactional memory", In *ACM Sigplan Notices,* October 2006, Vol. 41, No. 11, pp. 336-346.

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

[5] M. Abadi, T. Harris, and M. Mehrara, "Transactional Memory with Strong Atomicity Using Off-the-Shelf Memory Protection Hardware", In *14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, February 2009, pp.185-196.

[6] P. Felber, C. Fetzer, P. Marlier, and T. Riegel, "Time-based Software Transactional Memory", In *IEEE Transactions on Parallel and Distributed Systems*, December 2010, Vol. 21, Issue 12, pp. 1793-1807.

[7] D. Dice, Y. Lev, M. Moir, and D. Nussbaum, "Early Experience with a Commercial Hardware Transactional Memory Implementation", In *14th International Conference on Architectural Support for Programming Languages and Operating Systems*, March 2009, pp.157-168.

[8] V. J. Marathe, W. N. Scherer III, and M. L. Scott, "Adaptive Software Transactional Memory", In *19th International Symposium on Distributed Computing*, September 2005, pp.354-368.

[9] P. Felber, C. Fetzer, and T. Riegel, "Dynamic Performance Tuning of Word-Based Software Transactional Memory", In *13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, February 2008, pp.237-246.

[10] S. Goldberger, "Best Linear Unbiased Prediction In the Generalized Linear Regression Model", In *Journal of the American Statistical Association*, Volume 57, Issue 298, pp.369-375, 1962.

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

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

[13] B. David, "Programming with POSIX threads", Addison-Wesley Professional, ISBN-0201633922, 1997.

[14] A. Porfirio, A. Pellegrini, P. Sanzo, and F. Quaglia, "Transparent support for partial rollback in software transactional memories", In *Euro-Par 2013 Parallel Processing*, 2013, pp. 583-594.

[15] R. Adl-Tabatabai, T. Lewis, V. Menon, R. Murphy, B. Saha, and T. Shpeisman, "Compiler and runtime support for efficient software transactional memory", In *ACM SIGPLAN Notices*, June 2006, pp. 26-37.

[16] B. Saha, R. Adl-Tabatabai, L. Hudson, C. Minh, and B. Hertzberg, "McRT-STM: a high performance software transactional memory system for a multi-core runtime", In *ACM SIGPLAN symposium on Principles and practice of parallel programming*, March 2006, pp. 187-197.

[17] H.Avni, and N. Shavit, "Maintaining consistent transactional states without a global clock", In *Structural Information and Communication Complexity*, 2008, pp. 131-140.

[18] C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun, "STAMP: Stanford transactional applications for multi-processing". In *Workload Characterization. IISWC 2008. IEEE International Symposium* ,2008 September, pp. 35-46.

[19] C. Wang, W. Y. Chen, Y. Wu, B. Saha, and R. Adl-Tabatabai, "Code generation and optimization for transactional memory constructs in an unmanaged language", In *International Symposium on Code Generation and Optimization*, March 2007, pp. 34-48.

[20] H. Rito, and J. Cachopo, "ProPS: A Progressively Pessimistic Scheduler for Software Transactional Memory", In *Euro-Par 2014 Parallel Processing*, 2014, pp. 150-161.

[21] R. M. Yoo, & H. H. S. Lee, "Adaptive transaction scheduling for transactional memory systems", In *annual symposium on Parallelism in algorithms and architectures*, June 2008, pp. 169-178.

[22] P.Mathias, and T. R. Gross, "Performance evaluation of adaptivity in software transactional memory", In *Performance Analysis of Systems and Software (ISPASS)*, April 2011, pp. 165-174.

[23] S. Hong, T. Oguntebi, J. Casper, N. Bronson, C. Kozyrakis, and K. Olukotun, "EigenBench: A simple exploration tool for orthogonal TM characteristics", In *Workload Characterization (IISWC)*, 2010 December, pp. 1-1.

[24] M. Castro, L. F. W. Goes, C. P. Ribeiro, M. Cole, M. Cintra, and J. F. Mehaut, "A machine learning-based approach for thread mapping on transactional memory applications", In *High Performance Computing (HiPC)*, December 2011, pages 1-10.

[25] J. R. Quinlan, "Induction of Decision Trees", In *Machine Learn*, March 1986, pp.81-106.

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

[27] D. Rughetti, P. Sanzo, B. Ciciani, and F. Quaglia, "Machine learning-based self-adjusting concurrency in software transactional memory systems". In *Modeling, Analysis & Simulation of Computer and Telecommunication Systems*, 2012 August, pp. 278-285.

[28] N. Dong, N. A.Smith, and C. P. Rosé, "Author age prediction from text using linear regression", In *5th ACL-HLT Workshop on Language Technology for Cultural Heritage*, June 2011, pp. 115-123.

[29] Google Inc. Quantifying Movie Magic with Google Search. June 2013.

[30] N. R.Draper, H. Smith, and E. Pownell, "Applied regression analysis", Wiley, ISBN-0471170828, 1966.

[31] J. R. Quinlan, "Induction of Decision Tree", In *Machine learning*, 1986, pp. 81-106.

[32] P. Kemal, and S. Güneş, "A novel hybrid intelligent method based on C4. 5 decision tree classifier and one-against-all approach for multi-class classification problems", In *Expert Systems with Applications*, 2009, pp. 1587-1592.

[33] P. Xu, F. Davoine, H. Zha, and T. Denoeux, "Evidential calibration of binary SVM classifiers", In *International Journal of Approximate Reasoning*, 2015.

[34] Y. Freund, R. Schapire, and N. Abe, "A short introduction to boosting", In *Journal-Japanese Society for Artificial Intelligence*, 1999, Vol.*14*, pp.771-780.

[35] Daniel Fried, Zhen Li, Ali Jannesari, Felix Wolf, "Predicting Parallelization of Sequential Programs Using Supervised Learning"， In *12th IEEE International Conference on Machine Learning and Applications (ICMLA)*, December 2013, pages 72-77.

[36] Y. Freund, and L.Mason, "The alternating decision tree learning algorithm". In *ICML*, June 1999, pp. 124-133.

[37] K.Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, and K. A. Yelick, "The landscape of parallel computing research: A view from berkeley", In Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley. 2006.

[38] P. Colella, "Defining Software Requirements for Scientific Computing", presentation, 2004.

[39] L. Dagum, and R. Enon, "OpenMP: an industry standard API for shared-memory programming", In *Computational Science & Engineering*, 1998, pp. 46-55.

[40] N. H. Nie, D. H. Bent, and C. H. Hull, " *SPSS: Statistical package for the social sciences* ", In *New York: McGraw-Hill*, 1977.

[41] C. C. Chang, and C. J. Lin, "LIBSVM: A library for support vector machines", In *ACM Transactions on Intelligent Systems and Technology (TIST)*, 2011, Vol.2, issue 3, No. 27.