View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Lehigh University: Lehigh Preserve

Lehigh University
Lehigh Preserve

Theses and Dissertations

2011

A Machine Learning Approach to Adaptive Software Transaction Memory

Qingping Wang
Lehigh University

Follow this and additional works at: http://preserve.lehigh.edu/etd

Recommended Citation

Wang, Qingping, "A Machine Learning Approach to Adaptive Software Transaction Memory" (2011). *Theses and Dissertations.* Paper 1086.

This Thesis is brought to you for free and open access by Lehigh Preserve. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of Lehigh Preserve. For more information, please contact preserve@lehigh.edu.

A MACHINE LEARNING APPROACH TO ADAPTIVE SOFTWARE TRANSACTION MEMORY

by

Qingping Wang

A Thesis

Presented to the Graduate Committee

of Lehigh University

in Candidacy for the Degree of

Master of Science

in

Department of Computer Science and Engineering

Lehigh University

August 2011

© Copyright 2011 by Qingping Wang

All Rights Reserved

This thesis is accepted in partial
fulfillment of the requirements for the degree of
Master of Science.

_____
(Date)

_____
Michael Spear

_____

Acknowledgements

I am grateful to many people who have been a part of my graduate studies. First of all, I want to express my gratitude to my advisor Prof. Spear. He is the best advisor I could wish for. He is an insightful scholar, a helpful supervisor, a good teacher, an excellent colleague and a top hacker. His enthusiasm, inspiration and efforts spent on students helped me grow. Without his sound advice, I would have been lost.

My parents have always been the source of my strength. Without their support in my education, none of my successes would have been possible. It's their love that encouraged me to go thus far. Words cannot express my love for them.

I owe sincere gratitude to my labmate Sean. Your friendship was a source of great happiness, and you made this small town feel home to me. Thanks for your help that made my life easier many times.

Among other people, I want to thank my younger brother Jiping, as well as my best friends Yang and Zhe. You are always a part of my life.

Contents

List of Tables

List of Figures

Abstract

Since software transactional memory (STM) came into existence, numerous STM algorithms have been proposed. These algorithms differ in internal synchronization control mechanisms, and thus each is best for some class of workloads. In this thesis, we propose a system that applies machine learning techniques into an STM runtime system, so that the best algorithm can be selected at run time according to workload features. The performance achieved by this system outperforms any single STM algorithm, and approaches the best possible performance of policies proposed by STM experts.

Chapter 1

Introduction

1.1 STM overview

Software Transactional Memory(STM) is a concurrency control mechanism, analogous to the concept of *transactions* [24] used in database systems to control access to shared memory. With STM, code sections that require atomicity are annotated as transactions. Memory accesses inside a transaction are instrumented. Boundaries of transactions (places where transactions begin and end) are also instrumented. Instrumentation code for individual memory accesses and transaction boundaries resides in STM libraries, which are in charge of how concurrency is controlled. Accesses to shared memory succeed when a transaction commits, or all involved memory locations roll back to their previous states if conflicts with other transactions happen. A typical implementation of an atomic block is illustrated in Figure 1.1.

```
atomic{                 BEGIN_TXN
  X = 2 * X;            Write(&X, 2 * Read(&X))
}                       END_TXN
```

Figure 1.1: A typical atomic block using STM
The lefthand side corresponds to what the programmer would write. The righthand side shows how a compiler would transform the code

1.2 STM: An Abstraction over Synchronization Mechanisms

As a concurrency control mechanism, STM offers an alternative to traditional synchronization techniques, like locks, semaphores and monitors. With any of these, to achieve concurrency in programs, developers not only need to figure out critical sections that should be protected, but also they have to design a synchronization protocol to specify how they are protected. Overall, to write correct concurrent programs with those techniques, details in synchronization protocol implementation and correctness reasoning require a lot of developer effort, which can be tedious and error-prone.

Consider a simple banking system which has two accounts, a checking account and a savings account. There are two transactions, T_1 and T_2, that access and update those two accounts. Transaction T_1 transfers \$512 from checking into savings, and transaction T_2 transfers 16% of the balance of savings into checking. To guarantee correctness, the sum of balances in the accounts should remain the same after the transfer. Each transaction consists of non-atomic instructions that access and update accounts. We need to make sure that intermediate status of either account

```
Thread1                          Thread2
Lock(savings)                    Lock(savings)
Lock(checking)                   Lock(checking)
Checking = checking - 512        tmp = savings * 0.16
savings = savings + 512          saving = savings - tmp
Unlock(checking)                 checking = checking + tmp
Unlock(savings)                  Unlock(checking)
Unlock(savings)
```

Figure 1.2: A deadlock free solution to account transfer problem with 2PL

caused by one transaction is invisible to another transaction.

A possible solution to this problem using locks with a Two-Phase Locking (2PL) protocol [24] is described in Figure 1.2.

The problem with this is that 2PL may suffer from deadlock [30], where each thread is waiting for resources held by other threads, and thus no thread can make progresses. Careless Lock/Unlock placement will easily give rise to the deadlock problem. Figure 1.3 shows an example, where $Thread1$ and $Thread2$ are waiting for a lock held by each other and cannot make progress.

A solution to the account transfer problem using STM is illustrated in Figure 1.4, where no lock instruction appears anywhere, and thus no correctness reasoning is required on how locks should be used. The code looks much more succinct and it's not very different from writing a sequential program. This succinctness means that, when using STM, application developers only need to identify critical sections of programs, and annotate them as transactions using APIs exposed by STM libraries. STM libraries will automatically handle underlying synchronization issues at run time. Therefore, instead of spending a considerable amount of effort in implementing and debugging the synchronization protocol, application developers can focus on

```
Thread1                          Thread2
Lock(checking)                   Lock(savings)
checking = checking - 512        tmp = savings * 0.16
Lock(savings)                    savings = savings -tmp
savings = savings + 512          Lock(checking)
Unlock(checking)                 checking = checking + tmp;
Unlock(savings)                  Unlock(checking)
Unlock(savings)
```

Figure 1.3: A solution to the account transfer problem using 2PL with deadlock Thread 1 and Thread 2 enter into a mutual waiting situation if they simultaneously acquire their first lock and then try to acquire the second one

```
atomic {                              atomic{
    checking = checking - 512;            temp = checking * 0.16;
    savings = savings + 512;              checking = checking - temp;
  }                                       saving = saving + temp;
                                      }
```

Figure 1.4: Account Transfer with STM

application design and implementation. In Figure 1.4, there is no need to worry about deadlock since we do not place any locks in the code. The STM will handle synchronization instead.

To a certain extent, STM offers a higher-level of abstraction on concurrency control. STM library developers focus on implementations of various synchronization protocols, while application developers, that is, STM library users, only need to concentrate on application design and implementation.

1.3 Issues with STM

STM is a promising solution to shared memory synchronization. It can reduce programmers' burden in writing concurrent programs. However, STM is not a panacea. Since the concept of STM came into existence, dozens of STM algorithms have been proposed [9, 10]. Not surprisingly, none of them can outperform all others on all workload across various software and hardware platforms.

On the one hand, some STM algorithms can not be applied to some classes of workloads. For instance, SwissTM [11] can not be used to instrument workloads that require privatization-safety [27], even if it is a good choice on workloads requiring less strict semantics.

On the other hand, even if an algorithm does not suffer from applicability restrictions, it may not be the peak performer on a workload. For instance, for workloads featuring tiny transactions, Nano, a high-overhead and low-bottleneck variant of WSTM [13], is a premier choice; while for read-dominated workloads, TML [7] is usually very good. Figure 1.5 and Figure 1.6 illustrate how different STM algorithms perform on workloads with different features.

1.3.1 STM semantics

STM semantics defines how transactions interact with non-transactional code. A clear and simple semantics for STM is valuable for programmers to understand the programming construct, and thus help debug program errors with programming tools.

If STM is used to protect every access to shared memory data, it's expected to

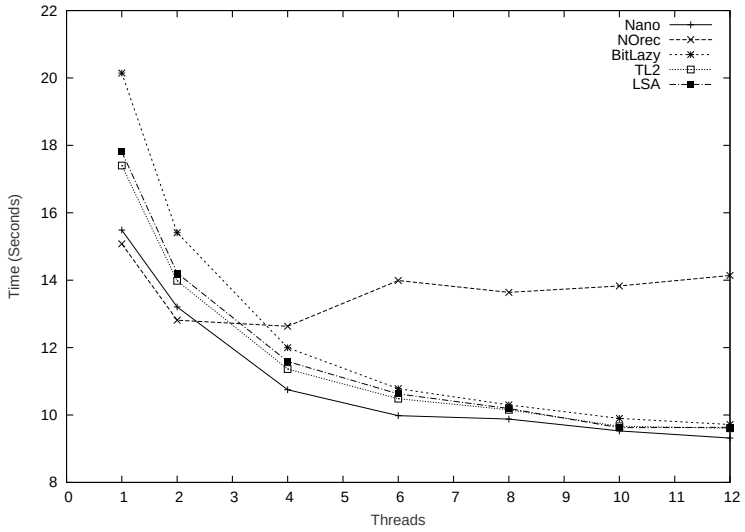

Figure 1.5: Performance of the STAMP SSCA2 workload
for common STM algorthms. The Nano algorithm is asymptotically worse than every algorithm tested, yet it is best since the workload has tiny transactions

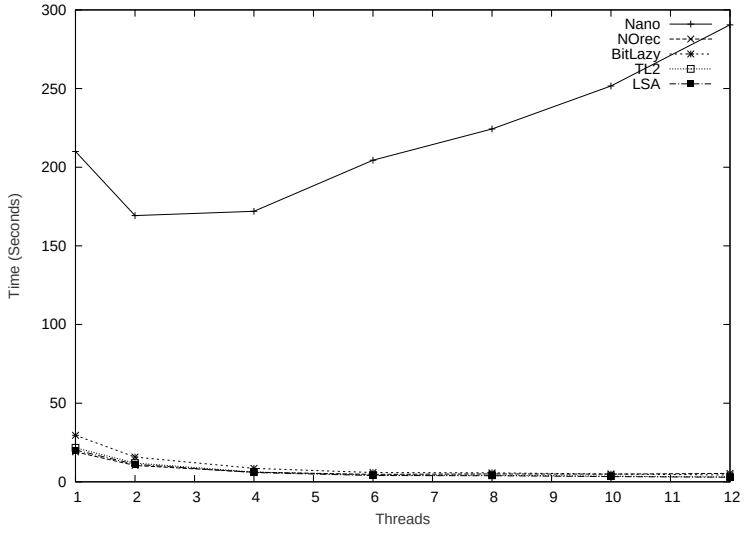

Figure 1.6: Performance of the STAMP Vacation workload
with low contention for common STM algorithms. Nano is the worst performer for Vacation with low contention

have the property called *strict serializaility* [24] from database systems: in every program execution, transactions would appear to occur, each atomically, in a global total order that is consistent with program order in every thread. This is the so called *Single Lock Atomicity*(SLA) [18], since it appears as if all transactions were protected by a single, system-wide mutual exclusion lock, so at most one block is in execution at a time. In SLA, shared data can be accessed non-transactionally as long as an equivalent lock-based program would be race-free. *Encounter-time Lock Atomicity*(ELA) [18] specifies that the compiler cannot reorder code within a transaction if the reorder could introduce a race in the equivalent lock-based code.

However, STMs could incur nontrivial cost to support *SLA* or *ELA*. For historical reasons, some STMs failed to provide SLA semantics [18], either because the target workloads only access shared memory with transactions; or because the need for *SLA* was not know when the algorithm was proposed. Thus, semantics provided by such algorithms are in conflict with *SLA*. When using these algorithms, there exist situations where shared data can be accessed by transactional code and non-transactional code simultaneously, which can violate correctness.

1.3.2 Implementation Strategies

Instrumentation for individual accesses and transaction boundaries is usually implemented in a library. The library provides concurrency control by mapping individual locations in memory to some form of metadata. Synchronization is enforced by operating on these metadata according to a single writer, multi-reader protocol. The library also detects deadlock and handles rollback.

Take TLRW [10] as an expample to illustrate how an STM works practically.

In TLRW, a data structure called *byte-lock* is used as synchronization metadata. A byte-lock tracks up to 56 concurrent readers and one writer. To write an address m, a thread i first tries to set the writer field of m's byte-lock to i; and if thread i is already a reader on m, it clears its read status from m. Then it spins until all other readers clear their read status, that is, all readers drain out from m. To read a location m, the value is simply returned if a thread i is already a reader or the writer on m; Otherwise, the thread registers itself as a reader of m, and only if m has no writer. If there is a writer, the thread uses back off before trying again. After a fixed number of tries, thread i aborts to prevent deadlock. By using this read/write locking approach internally, the algorithm retains correctness while shielding the programmer from the need to reason about locks.

There are a variety of STM algorithms in literature featuring different forms of metadata and synchronization protocols. Each of them works well for a class of workloads. Mainstream design and implementation strategies are described below.

Single Mutex Some STM algorithms use a global mutual exclusion lock to protect all transactions. In such implementations, write logging may be used to support self-abort (log old values of writing locations and write them back when the transaction aborts); or a reader-writer lock could be used if transactions with read-only operations dominate (this increases concurrency for readers).

Ownership Records (Orecs) In STM algorithms, such as TL2 [9], memory locations are mapped to a large table of ownership records (versioned locks). Reads do not modify locks but record lock versions; Writes acquire the lock either on the first encounter (eager), or at commit time (lazy). Eager systems usually implement in-place update of locations and undo logging, while lazy systems use buffered update

and redo logging. Moreover, orec systems usually employ a global shared counter to reduce overheads [9, 22], but the global counter can become a bottleneck when small writer transactions dominate, since it needs to be updated by every writer. Overall, Orec systems are typically good for workloads with large transactions, and usually scale well due to fewer bottlenecks than other systems. However, each write requires a costly compare-and-swap (CAS) operation.

Signatures In these algorithms, a transaction's accesses are represented as bit-vectors, or signatures, such that conflicts can be detected using fast vector intersection operations. Such STMs, like RingSTM [28], are livelock-free and do not require multiple CAS operations[28, 9]. These STMs are compatible with stronger language-level semantics by default. However, they suffer from larger bottlenecks and limited granularity of conflict detection.

Values There are STMs, e.g. NOrec [8], that do not use per-location metadata, instead logging all address/value pairs read. Conflicts are detected by checking if the values of reading locations have changed [21, 8]; and a single lock is used to protect commit operations. Algorithms in this style are livelock-free, and tend to have very low single-thread latency since no global metadata is maintained. They also provide very strong language-level semantics. However, such algorithms are not a good fit for workloads with frequent writer transactions, since the single lock limits performance, and checking for conflicts in large transactions can be expensive.

Bit and Byte Locks All of the designs described above use optimistic read mechanisms, such that no transaction can identify when it is accessing locations that another transaction is reading. In effect, readers are invisible to other transactions. Relatively cheap visible reader implementations have been achieved by maintaining

either bitmaps [17, 20] or wider bytelocks [10]. Reader visibility increases latency (due to read registration) and can result in more contention for metadata, but it simplifies conflict detection and resolution, and enables good semantics (ELA). TLRW [10] is a system in this class.

Read-Parallel Designs A few STM designs target specific domains, such as the eager and lazy variants of the TML algorithm [7, 8]. These provide extremely low latency for individual transactions due to the simplicity of metadata, but only allow either a set of readers or a single writer to execute at any time.

Chapter 2

Adaptive STM System

2.1 Motivation

That one STM algorithm fits well for all workloads is not true. There are good chances that the algorithm used to instrument a workload falls into pathologies, or performs poorly. Definitely, we want our system to be smart enough to avoid pathologies. To go one step further, even if there is no pathology, there could be chances of achieving better performance, and we want our system to be able to explore such possibilities.

To solidify our motivation, consider a program whose behavior is input dependent, and the execution exhibits phases (program sections with different characteristics), where the ideal algorithm varies from phase to phase. In this scenario, we desire that the STM library can dynamically select the best algorithm to instrument each phase, in order to maximize the performance. Therefore, to be able to achieve peak performance all the time, pathology avoidance and continuous exploration

(periodical exploration for better STM algorithms) are necessary.

2.2 Previous Adaptive STM Systems

Adaptive STM is not a new idea. There are previous works that sought to prevent pathologies, or to maximize performance using ad-hoc policies. The following are the most relevant works.

Worst-Case Progress Almost all current TM systems allow the runtime to automatically abort any transaction (by undoing the effects of its actions) if it conflicts with another transactions. This mechanism precludes execution of irrevocable actions, such as *I/O* and *system calls*, whose effects cannot generally be rolled back. Thus, it limits the application of transactions. Welc et al. [32] proposed a mechanism called single-owner read locks that supported irrevocable transactions. This mechanism allowed transition of a regular transaction into an irrevocable state on-the-fly during its execution. At any time, only one active irrevocable transaction can exist in the system, and it is guaranteed to commit since its revocation is prevented. Besides supporting *I/O* in transactions, systems like this can be used to guarantee progress, by guaranteeing that some transaction always commits.

Location-Level Adaptivity Sonmez et al. [25] investigated feedback-directed dynamic selection between different implementations of atomic blocks. In their strategy, atomic blocks were executed using STM with optimistic concurrency control, i.e. invisible reads; and variables that cause large numbers of aborts were dynamically identified. These *hot variables* were selectively switched to pessimistic concurrency control (visible reads). In this way, transactions that abort often are

deferred until they can complete. This strategy reduced single-threaded throughput (due to the cost of pessimistic concurrency control), but saved work wasted in aborted transactions. Moreover, it prevented pathologies at the cost of bringing overhead on hot variables detection.

Scalable Progress Guarantees Ni et al. [20] proposed a high-performance STM library which implemented multiple execution modes and a novel record based STM algorithm that supported both optimistic and pessimistic concurrency control. In their system, besides supporting irrevocable mode, the runtime was able to switch a transaction dynamically to obstinate mode, that is, a transaction can run concurrently with regular transactions, but has a higher conflict resolution priority. This system employed a novel indirection based interface to prevent overhead while supporting these mechanisms, and it was able to also avoid global coordination when switching the mode of a transaction.

Performance Via Feature Monitoring Marathe et al. introduced an adaptive STM system[16] that tracked a workload using a special API call named *early release*, which removes a location from the transaction's read log. If a workload used early release, locations would be locked at commit time. This technique improved throughput and lowered latency for transactions, but relied on the use of an uncommon feature.

Re-Parameterizing the STM Felber et al. [12] proposed TinySTM, a word-based STM implementation that used locks to protect shared memory location, and they performed dynamic tuning with it. In their system, one of the most important parameters that affected system performance was the number of locks used for concurrency control: increasing number of locks could reduce false sharing; while a

smaller number of locks could in principle reduce the validation time of an update transaction (since fewer locks are checked), at the cost of some performance penalty due to false sharing. To find the best number of locks needed, they periodically adapted the parameter at runtime, using a hill climbing algorithm.

Phased Execution PhTM [14] switched between hardware and software modes on a machine with hardware TM support. Events including the presence of transactions that were not supported by the hardware, excessive consecutive aborts and periodic timers caused the system to switch modes. PhTM supported switching between different *phases*, each implemented by a different form of transactional memory support, so the runtime could adapt between a variety of different transactional memory implementations according to the current environment and workload. However, the focus of this work was on hardware/software interaction, and PhTM did not consider switching among STM implementations, except for avoiding pathologies. In addition, some variants required shared-memory communication at the beginning of some transactions even when there was no mode switch in progress, which could act as a bottleneck.

Selecting Locks or Transactions Usui et al. [31] employed a combination of static and dynamic analysis to identify workloads for which locks outperformed STM, even when multiple threads were available. Clearly at one thread, a lock-based runtime with a lower latency is better. Additionally, if transaction latency is too high, and the cost of a lock moving between processors' caches is low, then at higher thread counts, the concurrency afforded by STM may not be worth its cost.

Pathology Avoidance The latest version of RSTM [26] supports adaptivity among different STM algorithms by combining the ideas from PhTM [14] with the

indirection-based interface of Ni et al. [20]. The system selects from 10 algorithms, to react to bad performance. Decisions are based on an algorithm's likelihood of pathology and precision of conflict detection.

Previous works on adaptivity support in STM lack generality. There are two common features of the approaches described above. First, decisions about which algorithm to use follow a statically specified policy, and usually only include a small set of options (or parameterized versions of a single algorithm). Second, the inputs to the adaptivity policy are based on very small feature set. These techniques are effective at improving performance and preventing pathologies, but none of them is able to maximize a program's performance by identifying the best STM algorithm for a dynamic program phase.

2.3 The Machine Learning Approach

Machine learning (ML) techniques are usually suitable for a complex problem with a huge search space that is hard to model. Given the complex diversity of features displayed by parallel programs and the subsequent amount of computing required to build models for program behaviors, it is hard to directly correlate a program feature with complex program behaviors. ML is a good candidate approach to such problems.

ML has been widely used in previous research for efficiently selecting compiler optimization parameters [6, 5, 2, 33], finding the best values for transformation parameters [19, 29, 4], and choosing the best algorithm to use for a sequential task [15].

We built an ML-based adaptive runtime system to improve the performance of STM-based programs. We implemented Case Based Reasoning(CBR) [1] in our general purpose framework. Our system can select among a broad set of TM algorithms during execution, to select the algorithm most likely to maximize the performance of the program.

Chapter 3

System Model

3.1 System Overview

To implement adaptivity in our STM library, we extended the adaptive version of RSTM [26]. Figure 3.1 depicts the components of the framework.

The adaptive STM system primarily consists of the offline training component, dynamic profiling component("more profiles" and "instrumented transaction" in Figure 3.1) and the online decision component("transaction profile" and "adaptive

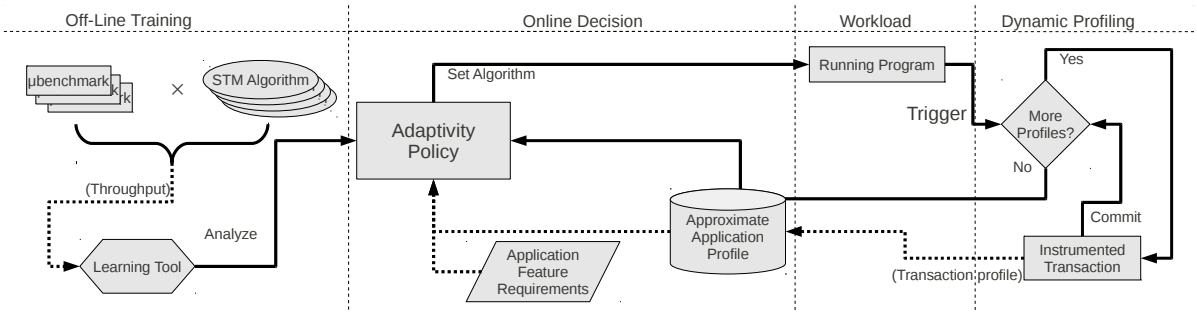

Figure 3.1: The System Overview

policy" in Figure 3.1), which are responsible for selecting the best algorithm to instrument the current phase of the running workload.

When the system starts, it first initializes itself. A dozen algorithms are registered in the system as candidates for dynamic selection. In addition, information obtained by offline training described in Chapter 4.2.1 is read by the system to create the adaptivity policy. Environment variables are read to specify if ELA semantics are required and how many transactions to run to profile the workload. After this configuration, a default STM algorithm is selected and the workload starts running.

While the workload is running, lightweight measurements detect the possibility that a new algorithm should be selected. We use the number of consecutive aborts for non-Mutex based algorithms, or for Mutex based algorithms the number of CPU cycles a thread waits before it can start a new transaction. We also track the total number of transactions committed by a specific thread. These metrics are intuitive, since too many aborts and too long of a waiting time indicate that starvation or livelock is possible. By checking the total commits in a specific thread, we can periodically sample after fixed commit counts $16^0, 16^1, 16^2, 16^3, k \times 16^4$, for all $k > 0$.

The metrics are checked every time when a transaction aborts. If the number reaches its threshhold, the system tries to switch to another algorithm by the on-line decision system. To make online decision, the system first switches to a ProfileTM algorithm, which is described in Chapter 4.1.1. ProfileTM runs in single-thread mode, and records a set of program characteristics of the workload. In ProfileTM mode, the excution of the workload does not stop. The amount of work ProfileTM

does varies from one to multiple transactions. This number is defined by an environment variable during system initialization.

When ProfileTM finishes its work, the system feeds the collected information to the decision component, which returns the algorithm most likely to maximize the performance according to the decision algorithm. The system switches from ProfileTM and instruments the workload with the algorithm returned by the decision component, and the system goes back to multithreaded mode. This process will repeat through the execution of the workload. Since the performance is being monitored at all times, as long as the system detects that the current instrumenting algorithm is not good, it will start the profiling process and try to find a better algorithm.

3.2 Discussion of the Framework

New STM algorithms can be easily integrated into the system. Every STM algorithm has an independent implementation, and they are registered in the system as function pointers. If there is any new algorithm proposed in the future and we want it to instrument workloads, what we need to do is add the implementation file, register its functions to the system, and retain the adaptive policy.

The decision component is also independent of other components in the system. Thus adding a new decision algorithm would not affect other components. To integrate a new decision algorithm, besides the implementation of the algorithm itself, we need to prepare the corresponding offline training component, which is completely isolated from others.

Like previous adaptive systems, our system can detect pathologies and guarantee progress. As described above, when the system detects livelock or starvation, it will switch to a better algorithm based on its adaptivity policy. To go one step further, just as one STM algorithm cannot excel on different workloads, the same algorithm might not have excellent performance for all phases of a workload. Our system is able to switch to a better algorithm for a different program phase. Moreover, during the whole process of workload execution, our system keeps exploring new candidates that can bring in better performance with reasonable overhead. Therefore, if there exists a better choice of algorithm, we have the chance to find it.

Chapter 4

System Implementation

In Chapter 3, we discussed how the system works. We have the offline training system, the profile component and the on-line decision algorithm. In this Chapter, we discuss the details of each component. In Chapter 4.1, representative program behaviors relevant to performance of STM algorithms are discussed, as well as the way we obtain that information. In Chapter 4.2, workloads with those representative program behaviors and the strategy to train the learning system on workloads are presented. Once the system trained is trained, details of how the system makes decisions based on application profiles are covered in Chapter 4.3.

4.1 Characterizing Workloads

Any system involving learning, either supervised or unsupervised, needs to identify a class of features of the system in order to generate rules that could be applied to it. A system that selects an appropriate STM algorithm for a specific workload must have some description of the workload behavior that provides a reliable basis for

decision making. As discussed in Chapter 2, previous approaches used a variety of measures to approximate the behavior of workloads, and many of them suffer from a lack of generality. To be general-purpose, we attack this problem via dynamic profiling to collect program behaviors that influence the performance of different STM algorithms.

4.1.1 Dynamic Features

The dynamic behavior of a program is measured via a combination of two techniques in our system. First, we use lightweight instrumentation on every transaction boundary to measure program-wide properties. Second, we developed a simple STM called ProfileTM, which is used to sample per-transaction program characteristics.

Boundary Instrumentation In the abort function of each STM algorithm, we update a per-thread counter of consecutive aborts. In the begin function, a counter records the number of ticks a thread waited before it started a new transaction. In the commit function, we update a per-thread count of committed writing transactions and read-only transactions. These numbers are queried when the workload read-only ratio (RORatio) is needed. In addition, a counter storing the value of the hardware tick counter is stored in the commit function. After every commit, its value is updated, while before a transaction begins, we subtract that value from the current hardware tick counter and add the difference to a per-thread accumulator. By dividing by the number of transactions, we can estimate the amount of non-transactional work (NonTxWork) between transactions.

ProfileTM Measuring program properties by whole program profiling is expensive (more than 5% slowdown). As it is reasonable that profiling should not incur overhead when not in use, we use sampling instead. Whenever the system detects that the current STM algorithm is not performing well, ProfileTM is used to instrument the running workload for a while (from one to several transactions). In ProfileTM, a fair ticket lock guarantees that only one transaction is running at a time, so there is no concurrency in the system, and every thread has a chance to be sampled. ProfileTM transactions sample the hardware tick counter when they begin, and again when they commit, to provide an estimate of the time that a transaction takes (TxTime). These transactions buffer all writes until commit time, and thus some reads must perform a lookup in the buffer. No global metadata is required by the buffer, which ensures compatibility with code that uses self-abort, and prevents possible races between self-aborting transactions and non-transactional code [23].

There are five distinguishable types of shared memory accesses in STM implementations. Each of them invokes overhead different than others: stores to new locations (*Writes*), write-after-write stores (*WAWWrites*), loads from a read-only context (*ROReads*), read-after-write loads (*RAWReads*), and loads from a non-read-only context that are not RAW (*RWReads*). Table 4.1 describes their differences with more detail. On every shared memory access, ProfileTM counts which of the five access types occurs. When the transaction commits, the entire dynamic profile is added to a log for use by the adaptive policy.

The cost to collect a dynamic profile with ProfileTM is low. First, boundary instrumentation that collects information to invoke ProfileTM is not expensive. Even though boundary instrumentation reads hardware counters and updates counters for

other events in every transaction, only a small number of load and store instructions are involved. Second, transactions have less latency under profile mode than under traditional STM algorithms, because ProfileTM runs in a single-threaded mode with no concurrency, so there is no cost for updating synchronization metadata. In addition, the number of profiling transactions is much smaller than normal transactions, and the frequency at which we incur adaptivity costs is not high. Moreover, ProfileTM guarantees progress.

4.2 Training

We extended the adaptive version of RSTM [26] to use the workload features described in Chapter 4.1 to select an STM algorithm during program execution.

4.2.1 OffLine Training Strategy

In previous literature, differences in the configuration of microbenchmarks led to different STM algorithms offering maximum throughput. These microbechmarks have representative program behaviors. We perform unsupervised off-line training in the learning component of our system. The training system is presented a set of microbenchmarks, a set of configurations of those benchmarks and a set of STM algorithms as input. From each microbenchmark-configuration-algorithm combination, it runs five 5-second experiments at different thread levels. The average throughput is recorded. Then it runs each microbenchmark-configuration combination using ProfileTM in single threaded mode to collect dynamic features of the

`ROReads`	These are reads performed by transaction T before its first write. ROReads typically have the lowest latency (in buffered-update STM systems, these reads do not require a write-set lookup)
`RWReads`	These are reads performed by T after it has performed at least one write. They always include the cost of a write-set lookup that does not succeed in buffered-update STM systems.
`RAWReads`	Reads to locations for which T has a speculative write often have low overhead. They appear as successful write-set lookup without further operations in buffered-update STM systems.
`Writes`	This is the number of distinct locations that have been written by T
`WAWWrites`	A write to a location that has already been written may have lower costs, since it may only need to update the buffered value
`NonTxWork`	When the gap between transactions is large relative to the duration of transactions, the best STM algorithm is typically one with low single-thread latency [31].
`RORatio`	For most STM algorithms, read-only transactions do not modify shared metadata. When the rate of writer transactions is low, these systems scale almost perfectly. As writers increase in frequency, the point at which read-only optimizations cease to be profitable varies with the STM algorithm.
`TxTime`	The average time a transaction takes.

Table 4.1: Dynamic workload features

workload. With these experiments, the system can figure out which STM algorithm a specific program may favor at a given thread level since it knows the best performer for the workload and the workload's characteristics. Then all the data is fed to the ML training policy to generate an adaptivity policy. The form of the policy is a data file that specifies the behavior of the decision system in different cases.

4.2.2 Offline Training workloads

In a production environment, it would be acceptable to tailor training data to the common-case for the target application. However, in order to show generality, we train using parameterized microbenchmarks instead, and thus measure what would serve as a lower bound on the effectiveness of our adaptive system. Our training workloads fall into the following categories.

Data Structure Traversals This class contains red-black trees, hash tables, and linked lists, with varying mixes of insert, lookup, and remove, and varying key ranges stored in the dataset. These workloads typically scale well, and correspond to the use of TM for creating concurrent data structures.

Pathology Test This workload causes livelock under eager acquisition, and starvation for most other STMs.

Overhead Finders These workloads expose overheads in the STM algorithm. Examples include shared counters, which highlight boundary latency, truly disjoint workloads, which show the cost of shared metadata on scalability, and read-sharing workloads, which emphasize the cost of visible reads.

Multiword Atomics These workloads use TM to perform multiword CAS operations of varying sizes, or to implement read N write 1 operations. We also created a

read N write N operation, to show how the order of reads and writes affects throughput.

Database Simulations These workloads aim to mirror more complex uses of transactions. In addition to various forest workloads (consisting of multiple operations on a set of red-black trees), we also provide a tree workload where every transaction performs writes.

As appropriate, we varied the non-transactional time between transactions, the number of locations accessed within a transaction, and the percentage of transactions that were read only. In total, this resulted in 213 different microbenchmark configurations, which we tested at many thread levels.

4.3 Adaptive Policies

In the system described in Chapter 3.2, any classification algorithm can be used to select an STM algorithm. The developer can easily create an adaptive policy completely independent of all other components. A completely automated ML system can generate the policy as the output of off-line training. At most, programmers need to offer some guidance when creating a policy with learning tools.

4.3.1 Expert Policies

These policies are written by a programmer, to satisfy arbitrary requirements. For example, RSTM (without ProfileTM and our dynamic adaptivity framework) provides expert policies to avoid pathology by transitioning the algorithm selection

according to a state machine. Our simplest expert policies capture the intuition that the best algorithm depends on the thread count. We provide three policies, depending on whether ELA semantics are required or not, and whether writers are expected to be frequent.

ThrX Assumes weak semantics are acceptable, and uses Mutex at 1 thread, and the LSA algorithm [12] otherwise. When ELA semantics [18] are not required, LSA is among the lowest latency and most scalable algorithms, unless contention is high.

ThrELA1 Provides ELA semantics, using Mutex at 1 thread and NOrec [8] otherwise. NOrec is among the most scalable STMs that provide strong semantics.

ThrELA2 Like ThrELA1, except for 8 or more threads, lazy TLRW [10] is used. TLRW has fewer bottlenecks than NOrec when writers are frequent.

4.3.2 ML-Based policies

We employ case based reasoning as the machine learning technique to automatically create an adaptive policy. It receives microbenchmark configurations and a set of STM algorithms; and outputs a data file that describes representative program behaviors and preferred STM algorithms as the adaptive policy.

Case Based Reasoning

Case-based reasoning is the process of solving new problems based on solutions of similar past problems. In case-based reasoning, a system creates a base containing configurations of environment, and the best responses of the system to the environment. In our system, our cases are program behaviors of microbenchmarks and the running environment (thread count), and the response to a case is the best STM

algorithm (the one with highest throughput). During program execution, the CBR policy scans the case base for entries that have the same number of threads as the workload; then it selects the entry that is most similar to the average of the collected transactional profiles, via a similarity metric, and returns the algorithm in that entry, that is the best performer for the program behavior for the microbenmark entry. Our CBR policies use the 8 dynamic features in Chapter 4.1.2. We combine the three read features into a single metric, and the two write features into another. We then consider all 31 possible combinations as candidate similarity metrics, using a normalized Manhattan distance. By retaining some metadata in the case base, we can always identify the training experiment that influenced a CBR decision, which aids in performance tuning.

Chapter 5

Evaluation

5.1 Test Environment

We built our STM codes and adaptive policies based on the adaptive version of RSTM [26]. The baseline adaptive RSTM provides 10 STM algorithms, and we added 9 more, which included published algorithms, new parameterizations of existing algorithms, and Nano (Chapter 1). All experiments were performed on an HP z600 with 6GB RAM and a 2.66GHz Intel Xeon X5650 (Nehalem) processor with six cores (12 hardware threads in total). Code was compiled with g++ version 4.5.1, in 32-bit mode with -O3 optimizations. All experiments are the average of 3 trials. We trained 6 versions of our adaptive policies: ELA refers to training conducted using only algorithms that provide Encounter-Time Lock Atomicity (ELA) semantics [18], and X refers to training on all 19 algorithms. We also considered three sets of training workloads: S1 used data structure traversals, pathology tests, and overhead finders. S2 used multiword atomics and database simulations. S1+S2

Name	Domain	Description	Tx Length	R/W Set	Tx Time	Contention
bayes	machine learning	Learns structure of a Bayesian network	Long	Large	High	High
genome	bioinformatics	Performs gene sequencing	Medium	Medium	High	Low
intruder	security	Detects network intrusions	Short	Medium	Medium	High
kmeans	data mining	Implements K-means clustering	Short	Small	Low	Low
labyrinth	engineering	Routes paths in maze	Long	Large	High	High
ssca2	scientific	Creates efficient graph representation	Short	Small	Low	Low
vacation	online transaction processing	Emulates travel reservation system	Medium	Medium	High	Low/Medium

Table 5.1: Descriptions of 9 STAMP benchmarks, adapted from [3]

used all training workloads.

We set an abort trigger at 16 consecutive aborts, a 2048-cycle loop spin on lock acquisition, and according to the commit thresholds described earlier. On any trigger, we collected a single transaction profile, as initial studies did not find a significant improvement in sample quality, but did observe noticeable slowdown in the Labyrinth workload, when collecting multiple profiles.

5.1.1 Evaluation Criteria

To evaluate our adaptive policies, we used the STAMP benchmark suite [3]. Details of the STAMP benchmarks are reported in Table 5.1. For the 9 recommended configurations, we tested each of the 19 STM algorithms at 1, 2, 4, 8, and 12 threads. Using this information, we created an Oracle dataset consisting of the best performer for each benchmark at each thread level. For each adaptivity policy, we tested each benchmark at each thread level, and computed its speedup versus the oracle (which is expected to be < 1). We scored each policy based on its per-benchmark harmonic mean speedup, as well as its STAMP-wide harmonic mean speedup. Occasional speedups > 1 occurred when the policy exploited program phases.

	Bayes	Genome	Intruder	KMeans (High)	KMeans (Low)	Labyrinth	SSCA2	Vacation (High)	Vacation (Low)	All
LSA	0.801	0.904	0.884	0.819	0.881	0.988	0.730	0.888	0.879	0.858
ThrX	0.803	0.936	0.979	0.892	0.917	0.995	0.785	0.974	0.959	0.910
CBRTime+RO	0.701	0.909	0.906	0.782	0.865	1.053	0.930	0.994	0.989	0.891

Table 5.2: Harmonic mean speedups on each STAMP benchmark, for the best single algorithm, the best configuration for the expert, and CBR adaptivity policies. In this table, there are no semantics requirements imposed on the policies. CBR is trained only on the S1 training set.

	Bayes	Genome	Intruder	KMeans (High)	KMeans (Low)	Labyrinth	SSCA2	Vacation (High)	Vacation (Low)	All
NOrec	0.877	0.911	0.889	0.645	0.720	0.992	0.555	0.877	0.883	0.791
ThrELA2	0.923	0.903	0.897	0.717	0.737	0.985	0.674	0.863	0.875	0.829
CBRRead	0.793	0.991	0.905	0.841	0.868	0.934	1.050	0.978	0.985	0.921

Table 5.3: Harmonic mean speedups when ELA semantics are required. CBR is trained only on the S1 training set.

5.2 Performance Summary: Preliminaries

Tables 5.2 and 5.3 list the best per-benchmark and STAMP wide harmonic mean speedups for each adaptive approach. Note that the oracle policy differs between the two tables, since ELA excludes LSA, TinySTM [12], TL2 [9], and Nano; consequently, quantitative comparisons cannot be made between tables. If only one algorithm can be used for all of STAMP, ELA favors NOrec while LSA is best otherwise. However, for several benchmarks this choice is far from ideal, resulting in a low 0.791 overall speedup for NOrec, and 0.858 for LSA. For X semantics, only TL2 was close (0.805); for ELA, TLRW variants, and orec variants, were close (above 0.73). The adaptivity policies included in RSTM perform poorly (not shown). These policies interpret transient high abort rates as pathology, and make permanent decisions toward fair but low-throughput algorithms. NOrec and LSA outperform the corresponding ELA and X RSTM policies.

5.3 Expert Policy Performance

The ThrX and ThrELA policies, which select an algorithm based only on the thread count, raise performance significantly. For ThrX, this improvement is completely due to avoiding overhead at 1 thread, as it chooses LSA otherwise. We recommend this approach without hesitation for any future STM design. However, ThrX still performs poorly on SSCA2, KMeans, and Bayes. ThrELA2, which chooses between Mutex, NOrec, and TLRW-lazy, is more nuanced. In choosing TLRW-lazy at 8 threads, it loses performance on Vacation. However, TLRW-lazy scales better than NOrec for small writing transactions, and in the end this improvement on KMeans and SSCA2 tips the scales in favor of ThrELA2 over ThrELA1 (which only uses Mutex and NOrec).

While we include Bayes performance in all of our evaluation, we are generally suspicious of this workload. The number and size of transactions run by each thread is dependent on the interleaving of a few transactions executed early in the workload; eager algorithms (particularly with visible reads) seem to deterministically choose a bad initial commit order, which can cause an order of magnitude slowdown. Similarly, a round-robin scheduling of transactions can occasionally cause a superlinear (> 4) speedup at 2 threads.

5.4 CBR Performance

We explored all combinations of 5 CBR feature categories, and considered all three training workloads. Given this large search space, we were able to find policies that offered exceptional performance on STAMP, With ELA semantics, the use of

a single feature, the read count of transactions, enabled a system that achieved a 0.921 speedup. This surpasses all other ELA adaptivity policies. With X semantics, our best performer only reached 0.891. Tables 5.2 and 5.3 only list the best CBR performer for a given semantics level. For reference, the CBRRead policy only had a 0.660 speedup under X semantics, and the CBRTime+RO policy achieved a 0.741 speedup under ELA (0.851 without Bayes). The most consistent CBR performer used two features: TxTime and NonTxWork (CBRTxTime+NonTxWork). For ELA semantics, it achieved a 0.892 speedup, and for X, a 0.890.

5.5 Impact of Training Data

Our CBR policies without exception performed best when trained only on the S1 training workloads. In considering the training workloads, S1 is drawn from STM microbenchmarks, whereas S2 is an attempt to model behaviors that we expect future TM programs to use. The explanation is simple: S2 contains many entries that, on a per-metric basis, are indistinguishable to our CBR similarity functions. Thus the S2 workloads can cause our policies to reject an otherwise valid choice of algorithm from S1, due to a similarity collision.

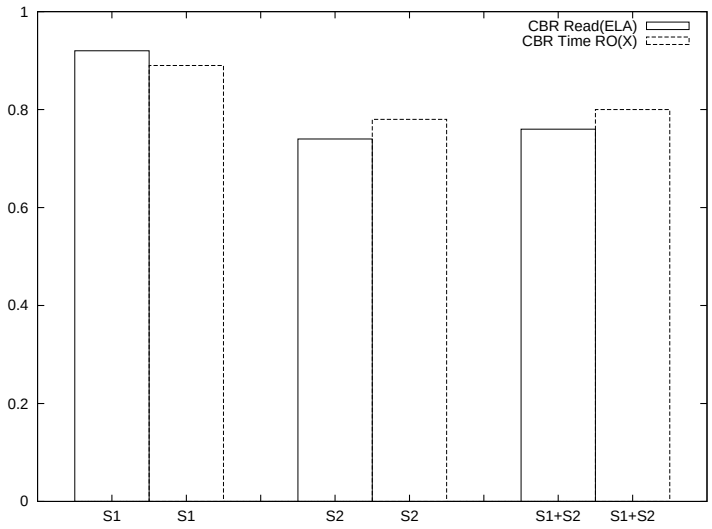

Figure 5.1: Impact of training data.
The best-performing CBR policies for ELA and X semantics degrade significantly when trained improperly.

Chapter 6

Conclusion

We believe that adaptive synchronization is necessary for high-performance shared memory programs. To that end, in this thesis, we introduced a low-overhead system for dynamically profiling the behavior of memory transactions. We also proposed an adaptivity mechanism based on machine learning that can exploit dynamic profiles to predict the STM algorithm that will maximize a workload's performance. By operating in this manner, our system is robust to program behaviors that are input-dependent, or that vary during distinct phases of execution.

As future work, we plan to investigate changes to our training strategy by exploring more metrics. In particular, we found that the ratio of transactional work to non-transactional had a strong impact on the choice of algorithm, yet our training workloads were not parameterized for non-transactional work. Similarly, the number of dynamic profiles to collect upon a trigger is crucial since the decision is largely based on it. We intend to learn this parameter automatically and explore more about its influence. In addtion, while our training workloads collect data about how

transactions behave in isolation, we did not explore metrics that consider the nature of concurrency in a program. More ML algorithms need to be considered to infer concurrency properties, such as metadata bottlenecks and conflict granularity.

In the longer term, we believe that many more questions will be easier to address given our results, our mechanisms, and our framework. Questions include adapting in response to other STM feature requests (such as I/O), adapting on architectures for which hardware TM support is available, and choosing among lock mechanisms for workloads that do not, or cannot, use transactions.

Bibliography

[1] Agnar Aamodt and Enric Plaza. Case-based reasoning: foundational issues, methodological variations, and system approaches. *AI Commun.*, 7:39–59, March 1994.

[2] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. F. P. O'Boyle, J. Thomson, M. Toussaint, and C. K. I. Williams. Using machine learning to focus iterative optimization. In *Proceedings of the International Symposium on Code Generation and Optimization*, 2006.

[3] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IISWC '08: Proceedings of The IEEE International Symposium on Workload Characterization*, 2008.

[4] John Cavazos and Michael F. P. O'Boyle. Automatic tuning of inlining heuristics. In *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, 2005.

[5] John Cavazos and Michael F. P. O'Boyle. Method-specific dynamic compilation using logistic regression. In *Proceedings of the 21st annual ACM SIGPLAN*

Conference on Object-oriented Programming Systems, Languages, and Applications, 2006.

[6] Keith D. Cooper, Alexander Grosul, Timothy J. Harvey, Steven Reeves, Devika Subramanian, Linda Torczon, and Todd Waterman. Acme: adaptive compilation made efficient. In *Proceedings of the 2005 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, 2005.

[7] Luke Dalessandro, Dave Dice, Michael Scott, Nir Shavit, and Michael Spear. Transactional mutex locks. In *Proceedings of the 16th International Euro-Par Conference on Parallel Processing: Part II*, 2010.

[8] Luke Dalessandro, Michael F. Spear, and Michael L. Scott. Norec: streamlining stm by abolishing ownership records. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2010.

[9] Dave Dice, Ori Shalev, and Nir Shavit. Transactional locking ii. In *In Proceedings of the 20th International Symposium on Distributed Computing*, 2006.

[10] Dave Dice and Nir Shavit. Tlrw: return of the read-write lock. In *Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures*, 2010.

[11] Aleksandar Dragojević, Rachid Guerraoui, and Michal Kapalka. Stretching transactional memory. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2009.

[12] Pascal Felber, Christof Fetzer, and Torvald Riegel. Dynamic performance tuning of word-based software transactional memory. In *Proceedings of the 13th*

ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), 2008.

[13] Tim Harris and Keir Fraser. Language support for lightweight transactions. In *Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, 2003.

[14] Yossi Lev, Mark Moir, and Dan Nussbaum. Phtm: Phased transactional memory. In *Proceedings of the 2nd ACM SIGPLAN Workshop on Transactional Computing*, 2007.

[15] Xiaoming Li, María Jesús Garzarán, and David Padua. A dynamically tuned sorting library. In *Proceedings of the international symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, 2004.

[16] Virendra J. Marathe, William N. Scherer III, and Michael L. Scott. Adaptive software transactional memory. In *Proceedings of the 19th International Symposium on Distributed Computing*, 2005.

[17] Virendra J. Marathe, Michael F. Spear, Christopher Heriot, Athul Acharya, David Eisenstat, William N. Scherer III, and Michael L. Scott. Lowering the overhead of nonblocking software transactional memory. In *Proceedings of the 1st ACM SIG-PLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, 2006.

[18] Vijay Menon, Steven Balensiefer, Tatiana Shpeisman, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Bratin Saha, and Adam Welc. Practical weak-atomicity

semantics for java stm. In *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, 2008.

[19] Antoine Monsifrot, François Bodin, and Rene Quiniou. A machine learning approach to automatic production of compiler heuristics. In *Proceedings of the 10th International Conference on Artificial Intelligence: Methodology, Systems, and Applications*, 2002.

[20] Yang Ni, Adam Welc, Ali-Reza Adl-Tabatabai, Moshe Bach, Sion Berkowits, James Cownie, Robert Geva, Sergey Kozhukow, Ravi Narayanaswamy, Jeffrey Olivier, Serguei Preis, Bratin Saha, Ady Tal, and Xinmin Tian. Design and implementation of transactional constructs for c/c++. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications*, 2008.

[21] Marek Olszewski, Jeremy Cutler, and J. Gregory Steffan. Judostm: A dynamic binary-rewriting approach to software transactional memory. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, 2007.

[22] Torvald Riegel, Pascal Felber, and Christof Fetzer. A lazy snapshot algorithm with eager validation. In *Proceedings of the 20th International Symposium on Distributed Computing*, 2006.

[23] Tatiana Shpeisman, Ali-Reza Adl-Tabatabai, Robert Geva, Yang Ni, and Adam Welc. Towards transactional memory semantics for c++. In *Proceedings of the*

twenty-first annual symposium on Parallelism in algorithms and architectures, 2009.

[24] Abraham Silberschatz, Henry Korth, and Sudarshan. *Database Systems Concepts*. McGraw-Hill, Inc., New York, NY, USA, 6 edition, 2006.

[25] Nehir Sonmez, Tim Harris, Adrian Cristal, Osman S. Unsal, and Mateo Valero. Taking the heat off transactions: Dynamic selection of pessimistic concurrency control. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, 2009.

[26] Michael F. Spear. Lightweight, robust adaptivity for software transactional memory. In *Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures*, SPAA '10, 2010.

[27] Michael F. Spear, Virendra J. Marathe, Luke Dalessandro, and Michael L. Scott. Privatization techniques for software transactional memory. In *Proceedings of the Twenty-sixth Annual ACM Symposium on Principles of Distributed Computing*, 2007.

[28] Michael F. Spear, Maged M. Michael, and Christoph von Praun. Ringstm: scalable transactions with a single atomic instruction. In *Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures*, 2008.

[29] Mark Stephenson and Saman Amarasinghe. Predicting unroll factors using supervised classification. In *Proceedings of the International Symposium on Code Generation and Optimization*, 2005.

[30] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2007.

[31] Takayuki Usui, Reimer Behrends, Jacob Evans, and Yannis Smaragdakis. Adaptive locks: Combining transactions and locks for efficient concurrency. In *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*, 2009.

[32] Adam Welc, Bratin Saha, and Ali-Reza Adl-Tabatabai. Irrevocable transactions and their applications. In *Proceedings of the twentieth annual Symposium on Parallelism in Algorithms and Architectures*, New York, NY, USA, 2008. ACM.

[33] Kamen Yotov, Xiaoming Li, Gang Ren, Michael Cibulskis, Gerald DeJong, Maria Garzaran, David Padua, Keshav Pingali, Pau l Stodghill, and Peng Wu. A comparison of empirical and model-driven optimization. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, 2003.

Vita

Qingping Wang was born to Xingen Wang and Lianmei Hu on Sep 24th, 1985 in Wuhan, China. He grew up and finished his education from primary school to college in his hometown. In June 2009, he recieved a bachelor's degree in Computer Science from Wuhan University, Wuhan, China. After that, he came to the U.S and attended graduate school at Lehigh University.