2016

Fairness and Approximation in Multi-version Transactional Memory.

Basem Ibrahim Assiri
Louisiana State University and Agricultural and Mechanical College

FAIRNESS AND APPROXIMATION IN MULTI-VERSION TRANSACTIONAL MEMORY

A Dissertation

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

in

The Department of Electrical Engineering & Computer Science
The Computer Science and Engineering Division

by
Basem Assiri
B.S., King Khalid University, 2007
M.S., University of Wollongong, 2009
August 2016

Acknowledgments

I would like to express my gratitude to my advisor and teacher Costas Busch for his advice, discussions, patience and effort. I am grateful to Maggie Edwards for her directions, discussions and insight. I am also thankful to my teachers at Louisiana State University: Rajgopal Kannan, Jian Zhang, Rahul Shah, Evangelos Triantaphyllou, Jianhua Chen, Gerald Baumgartner and Sonja Wiley. I would like to thank the Computer Science and Engineering Division at Louisiana State University for providing excellent and well prepared courses. I would like to thank my sponsor, Jazan University (in Saudi Arabia), for awarding me with a PhD scholarship. Finally, I would like to thank my parents and my wife for their encouragement and support.

Table of Contents

Acknowledgments ii

Abstract v

Chapter 1: Introduction 1
1.1 STM Scheduling 5
1.1.1 The Machine Learning Techniques 5
1.1.2 Fairness of Transactions' Scheduling Using Machine Learning Techniques 5
1.2 STM Approximation 6
1.2.1 Approximately Opaque Read-only Transactions 7
1.2.2 Approximated Opacity for Other Kinds of Transactions 7
1.3 The Garbage Collector 8
1.4 Outline of the Dissertation 8

Chapter 2: Related Work 10

Chapter 3: Notation and Definition 15
3.1 Kinds of Operations and Sequential Object 15
3.2 Approximated Opacity 17

Chapter 4: Transactional Memory Scheduling Using Machine Learning Techniques 24
4.1 Introduction 24
4.2 The Supervised Machine Learning Techniques and Dataset 24
4.2.1 Support Vector Machine SVM 27
4.2.2 K-Nearest Neighbor 28
4.3 The Unsupervised Machine Learning Model 29
4.4 The Design of our Algorithm 31
4.4.1 The Design of LSA 31
4.4.2 Scheduling Algorithm 32
4.5 Experimental Results and Discussion 33

Chapter 5: Approximately Opaque Multi-version Transactional Memory for Read-only Transactions 42
5.1 Introduction 42
5.2 Design of the Algorithm 43
5.3 Correctness of the Algorithm 46
5.4 Experimental Results 53

Chapter 6: Approximately Opaque Transactional Memory for Read/Write and Count Objects . . . 56
6.1 Introduction . . . 56
6.2 The Design of the Algorithm . . . 56
6.2.1 Read/Write Operations . . . 58
6.2.2 Counting Operations . . . 59
6.3 Correctness of the Algorithm . . . 60
6.4 Experimental Results . . . 70
6.5 Approximately Opaque Multi-version STM . . . 72
6.6 Multi-version STM Experimental Results . . . 74

Chapter 7: Approximately Opaque Transactional Memory for Queue Objects 76
7.1 Introduction . . . 76
7.2 The Design of the Algorithm . . . 76
7.3 Correctness of the Algorithm . . . 79
7.4 Experimental Results . . . 89

Chapter 8: Garbage Collector . . . 92
8.1 Introduction . . . 92
8.2 Design of the GB . . . 92
8.3 Correctness of the Algorithm . . . 93

Chapter 9: Conclusion . . . 95

References . . . 97

Appendix. Copyrights Letters . . . 101

Vita . . . 105

Abstract

Shared memory multi-core systems benefit from transactional memory implementations due to the inherent avoidance of deadlocks and progress guarantees. In this research, we examine how the system performance is affected by transaction fairness in scheduling and by the precision in consistency. We first explore the fairness aspect using a Lazy Snapshot (multi-version) Algorithm. The fairness of transactions scheduling aims to balance the load between read-only and update transactions. We implement a fairness mechanism based on machine learning techniques that improve fairness decisions according to the transaction execution history. Experimental analysis shows that the throughput of the Lazy Snapshot Algorithm is improved with machine learning support.

We also explore the impacts on performance of consistency relaxation. In transactional memory, correctness is typically proven with opacity which is a precise consistency property that requires a legal serialization of an execution such that transactions do not overlap (atomicity) and read instructions always return the most recent value (legality). In real systems there are situations where system delays do not allow precise consistency, such as in large scale applications, due to network or other time delays. Thus, we introduce here the notion of approximate consistency in transactional memory. We define K-opacity as a relaxed consistency property where transactions' read operations may return one of K most recent written values. In multi-version transactional memory, this allows to save a new object version once every K object updates, which has two benefits: (i) it reduces space requirements by a factor of K, and (ii) it reduces the number of aborts, since there is smaller chance for conflicts. In fact, we apply the concept of K-opacity on regular read and write, count and queue objects, which are common objects used

in typical concurrent programs. We provide formal correctness proofs and we also demonstrate the performance benefits of our approach with experimental analysis. We compare the performance of precise consistent execution (1-opaque) with different consistency values of K using micro benchmarks. The results show that increased relaxation of opacity gives higher throughput and decreases the aborts rate significantly.

Chapter 1
Introduction

Currently multi-processor systems have become essential in handling bursts of data usages on computers. One of the advanced techniques to enhance and improve quantity and quality of computation is parallel computing. Parallel computing means to run many computations and tasks at the same time [2]. However, some problems require access to memory to be solved. Hence, multi-processor systems need to be supported by efficient memory techniques to allow concurrent operations which increases the throughput and preserves the consistency. To clarify, consistency means that there are no conflicts and the state of the system after each action is predictable [22]. We can achieve memory consistency, if there are some rules to make the results of the operations predictable. For example, if $x = 1$, and there are two write operations running simultaneously where the first one writes $x = 2$ and the second writes $x = 3$, we must have rules that decide which operation executes first. Only then we will be able to predict the value of x after each operation.

In addition, an important way to deal with the difficulty of writing concurrent applications is to use Transactional Memory (TM) [19]. TM enhances systems performance because it allows to avoid locks cost and problems [25]. TM supports concurrent processing by allowing concurrent access to shared data. Also, Software Transactional Memory (STM) is now available to perform as efficiently as hardware one but it provides more flexibility and control over the concurrent programming [34]. STMs use the principle of shared memory transaction [19] which is a piece of code or a finite sequence of operations that access local and shared memory. The basic operations are either read or write. The read operation reads the data from

the memory while the write operation writes data to the memory. A transaction is called read-only transaction if it has only read operations, and is called update transaction if it has at least one write operation. When all operations are executed, a transaction commits or aborts. Commit means to save all the changes and effects which are made by the current transaction. Abort means to discard all actions and changes that are made by the transaction [34][19].

To process multi-tasks in parallel, the memory access requires appropriate techniques to avoid conflicts. Conflicts happen between concurrent transactions if one reads a shared memory object and the other writes to it, or if both write to the same object at the same time. Hence we have three kinds of conflicts which are read after write (rw), write after read (wr) and write after write (ww). To avoid such conflicts we need first to know the critical sections which are the parts of memory (shared objects) or the piece of program that may cause conflicts. In some cases we use mutual exclusion and lock-based techniques to assure the consistency. In mutual exclusion and lock-based techniques only one transaction is allowed to enter the critical section. Using locks is safer but it is very expensive and it may cause other problems such as deadlock and starvation. Deadlock happens when two transactions are blocking each other such that each transaction has a lock that is required by the other. Starvation happens when some transactions may wait forever and do not get a chance to execute. Therefore, precise lock-free synchronization is required. In fact, the isolation and atomicity of transactions support lock-free techniques as they allow for many transactions to access the same shared data and the concurrent execution appears as if it is sequential [19][15].

STMs run transactions concurrently, and then detect conflicts. A transaction aborts only if there is conflict that affects the correctness of the execution. High

rate of parallelism may lead to high rate of aborts resulting in huge negative impact on performance [6].

Many applications and algorithms avoid the abort of read-only transactions because those transactions do not directly affect the consistency of the memory. Perelman et al. [27], show that 80% of the execution time might be wasted as a result of aborting read-only transactions. In order to minimize aborts and maintain consistency, *multi-version* STMs keep multiple versions for each object in the memory [7]. In this way, when an update transaction commits, it creates new versions for every objects in its *write set*, which is a set of all object the transaction writes to. Using multiple versions of memory objects helps read-only transactions to successfully commit (without aborting at all) by preserving the versions in the *read sets* of the transactions, which is a set of all object the transaction reads. However, using multi-version STMs causes a surplus negative effect on the systems' space complexity.

In this research, we aim to raise the throughput of multi-version algorithms, decrease the aborts rate and reduce the negative impact on the space complexity. Actually, we suggest to increase the throughput by maintaining the scheduling of the transactions and by reducing the precision in consistency using the principal of approximation. First we examine the fairness of scheduling. It is obvious that algorithms work well on some systems and under some conditions but they fail with others. However, under any conditions transactions' scheduling has a huge impact on the performance as we can increase the throughput by scheduling the independent transactions to execute together. Also, some kinds of transactions, such as read-only ones, do not cause conflicts because they do not change the memory content. Therefore, they can be scheduled to execute concurrently. Clearly, different systems have different data structures and various transactions' structure,

length and number. Thus, we need to find the suitable scheduling procedure for each system. We use machine learning techniques to find out the suitable scheduling procedure based on the history of the transactions. Moreover, focusing only on the throughput may result in unfair scheduling for transactions as some transactions may usually delay. Hence, we acquire fair scheduling to prevent starvation.

On the other hand, relaxing opacity to avoid some aborts is sometimes necessary on non-sensitive data and non-sensitive systems, or where data changes frequently. In large scale network systems when an update happens to an object in a local memory, it takes some time to update the object's global view. In fact, some read operations might be executed on other local copies during such delay (between the updates of local and global memory) which makes them illegal and causes aborts. Also, long read-only transactions may cause many aborts as well.

In addition, in real life there are many kinds of systems that do not require to have precise results, especially on non-sensitive data [36][30]. For example some approximated results might be acceptable for Online Analytical Processing (OLAP) that answers nested and complicated database queries such as inventory queries and reports generations [9]. Also, approximated results work with Decision Support Systems such as queries about average income and the percentage of newborns in the country [1]. Moreover, social network mining systems result in huge data with very frequent changes which increases the importance of approximation [8]. Systems for advertising and recommendation (for movies, music or restaurants) could allow approximated results since this does not propagate any risks. Sometimes the approximation is part of the system specifications such as the systems that use sensors for temperatures, locations and weather forecasting.

In such cases, the approximated data is enough and that allows to relax the precision level. Therefore, we will be able to commit some transactions even if

they violate consistency which increases throughput and reduces aborts. Thus, we introduce K-consistency concept to relax the consistency, where K is an integer number that represents the approximation degree. It means if the result is within K approximation degree it commits; otherwise, it aborts.

Furthermore, we use approximation to reduce the number of created versions (or the number of updates to the global copies of the objects). As we mentioned before, using multi-version algorithms, each committed update transaction creates a new version for each object in its own write set. However, with approximation we create a new version for each object every K commits which improves the space complexity and reduces the total number of versions by factor K compared to STMs that do not provide approximated opacity.

1.1 STM Scheduling

1.1.1 The Machine Learning Techniques

At the beginning, we explain the machine learning models which are supervised and unsupervised learning models. In supervised learning, we generate a function that maps inputs to suitable outputs that is called labels or classes. Experts often provide some training examples that supply systems with labels or classes. For example, in classification problems, the learner suggests a function that maps vectors of inputs into classes by looking at training examples. On the other hand, unsupervised learning is to model a set of inputs while labels are unknown during training [18]. In fact, the model itself recognizes labels while it is running. In our problem we use supervised learning models, namely Support Vector Machine and K-Nearest Neighbor, and unsupervised learning with Hidden Markov Model.

1.1.2 Fairness of Transactions' Scheduling Using Machine Learning Techniques

In Chapter 4, we suggest to increase the throughput of a classic multi-version algorithm which is Lazy Snapshot Algorithm (LSA) by maintaining the scheduling

of the transactions. The fairness of transactions' scheduling is to balance between read-only transactions and update transactions. In addition, the duration of read-only and update transactions and the transactions' dependencies vary from one system to another, which implies that different scheduling is required for different systems. Therefore, fairness of scheduling is to decide how many read-only transactions to be committed per update ones and this ratio is called the Fairness Value (FV). The FV is determined according to a machine learning technique through keeping the track of transactions' history.

By recording a prefix of transactions' execution, we find out the order and number of read-only and update transactions and pass them to the learning model. Two supervised machine learning techniques which are the Support Vector Machine (SVM) and the K-Nearest Neighbor (KNN) are used for classification. According to the prefix information, SVM and KNN map the given information to the suitable FV. Furthermore, we compare the results of the supervised machine learning techniques with the Hidden Markov Model (HMM), which is an unsupervised machine learning technique. In fact, the KNN shows the best performance and accuracy. Also, the scheduling using KNN satisfies the fairness and improves the throughput of LSA.

1.2 STM Approximation

Correctness of concurrent execution in transactional memory is typically proven with *opacity* [15], which is a consistency property that requires a legal serialization of an execution such that transaction intervals do not overlap (atomicity of transactions) with preserving of real-time order of events, and read operations always return the most recent written value to the object (legality of objects). However, system applications may not permit precise consistency of an execution due to hardware delays (network delays, I/O operations, etc.) or other unforeseen reasons

during execution. In such cases it may be allowed for read operations to return old values.

We introduce the notion of *approximately opaque* multi-version permissive transactional memory, a novel STM algorithm that never aborts read-only transactions and reduces the number of newly created object versions. We create a new version with each K commits on the object.

We define K-*opacity* as a relaxation of the opacity correctness property where a transaction can read one of the last K committed values on the object (relaxation of legality).

1.2.1 Approximately Opaque Read-only Transactions

At the beginning, we apply K-opacity on read-only transactions, while update transactions access only the latest version of an object. In a read-only transaction, a read operation tries to first read the latest value of the object, and if that value violates the correctness of the execution, then it reads the value of a saved version of the object which allows it to commit. In contrast, update transaction must always read the latest value of the object.

1.2.2 Approximated Opacity for Other Kinds of Transactions

The promising results of applying the K-opacity concept on read-only transactions encourages us to apply it on update transactions as well. Actually, unbounded number of read operations can access each version of any object, but only K write operations can be committed on a version, and they can be concurrent.

Precisely, a read operation can read one of the last K writes on the prospective object. However, the committed update transactions may write some values based on those inconsistent reads which would make them produce inaccurate results. If some new transactions use those inaccurate results to write, the errors would propagate. To prevent error propagation, we do not allow to read from inconsistent

values. Although we can have up to K writes and they can be in parallel but only one of them creates a new version (or overwrites the object if we use single-version). Indeed, the one that creates the new version is definitely correct and consistent. Thus we have a consistent version every K writes and newer transactions only read from the consistent versions.

To guarantee that the created versions are consistent, we introduce the idea of a writer list to implement the concept of visible writes where we record the values of all committed write operations. These values help to achieve the correct value on the new version. Therefore, every system or data structure needs a specific way for recording and correcting. However, we only focus on the concurrent read, write, counting and queue operations.

1.3 The Garbage Collector

We provide a garbage collector to get rid of old versions. Our garbage collector does not remove the version that shows the last written value. Our garbage collector works on the old version lists of objects which contain the versions that are saved every K commits on the objects. Our garbage collector decides unwanted versions according to the needs of the live transactions. Certainly, it finds the live transaction with the smallest timestamp min_{ts}. Then, it deletes all old versions of the objects except the ones that are needed by min_{ts} or the ones that have greater timestamps. Obviously, we keep the versions that might still be required by live transactions.

1.4 Outline of the Dissertation

The rest of this dissertation is organized as follows: In Chapter 2 we discuss some related works. The system model, definitions and preliminaries are presented in Chapter 3. In Chapter 4, we present the design of the machine learning techniques, scheduling algorithm and some experimental results. Chapter 5 discusses

the approximation of read-only transactions. In Chapter 6, we introduce the approximation of all transactions that have read, write and counting operations, while Chapter 7 shows the approximation of transactions that execute queue operations. Chapter 8 discusses the design and the benefits of the garbage collection procedure. Chapter 9 concludes the research with some discussion and future works.

Chapter 2
Related Work

STM was introduced to support multi-processor and multicore systems [34]. However, it requires careful implementation to avoid conflicts. STMs allow many transactions to be executed in parallel and each transaction validates to either commit or abort. In fact, STM is a suitable paradigm to support parallel processing since it can guarantee progressiveness and permissiveness properties [34]. Progressiveness on STM is proposed to reduce aborts such that if there is conflict between transactions, at least one of the conflicted transactions commits [17]. Permissiveness also reduces aborts by avoiding the abortion of read-only transactions (which only have read operations) [34][5].

Devietti et al. [10] show how to acquire determinism and consistency in the execution. Transactions can be classified according to whether the objects' statuses are private or shared, and whether the operations are reads or writes. They offer different techniques to handle determinism. In some situations, thread must get the token to execute a transaction while in other situations thread can execute transaction directly. The token is used to guarantee sequential execution for conflicted transactions. However, this algorithm has two problems: deadlock and starvation. In our algorithm in Chapter 4, we schedule transactions according to the type of their operations, while in the other algorithms we use timestamp-based scheduling.

Another approach to reduce aborts is the multi-version permissive STMs, which keep multiple versions of each memory object [14][28]. In case of a conflict, this kind of STMs could prevent aborts by allowing some operations of the conflicted transactions to use old versions of the perspective object, thus maintaining the

consistency. The main quality of using multi-version permissive STMs is to never abort read-only transactions.

One example of the famous multi-version algorithms is LSA [29]. With LSA, we check the states of the consistency of each object version at the access moment (when transaction accesses the object to execute the operation). Therefore, we can build consistent snapshots during the execution of the transaction to assure that the transaction reads consistent versions and guarantees correctness of execution. The correctness of transaction's execution is verified if the snapshots of all objects' versions it accesses are consistent, which will be explained in detail in Chapter 4. However, we claim that the type, length and order of transactions affect the performance of any algorithm. Thus, we suggest scheduling transactions in a way that suits the transaction's operation types and number, as well as the data structure they use.

Moreover, one of the machine learning techniques used in our research is SVM. SVM is a supervised machine learning technique that enables programmer to design a dataset that includes some training examples. Then, SVM classifies the data according to the given examples. The accuracy of classification of SVM will be computed according to the margins and distances among classes [23][4].

Another supervised machine learning technique is K-Nearest Neighbor, which classifies object based on the nearest training examples [3][33]. In other words, it clusters similar data in classes. Both of them can be used in our algorithm to determine the proper FV.

A further idea presented by Wang suggests to use different algorithms according to the inputs' types [37]. Wang uses transactional memory with a machine learning model and compares it with the one that uses an expert system. However, the experiment focuses on some hardware features such as transactional memory type

and cache size. On the other hand, our algorithm uses a learning model to find the suitable FV to guide the scheduling process and improve the performance.

Another work applies Markov Chain to improve STM performance [11]. It uses the Markov Chain for scheduling to control the contention of transactions and decide on blocking temporarily when needed. In fact, they focus on contention and the number of transactions running in parallel regardless of the type or the length of these transactions. However, in our work we use the HMM which is an unsupervised learning model [21][11], to decide how to schedule the transactions based on their types.

In concurrent execution different strategies are used to track conflicts. The use of timestamps helps to track the order of operations such that each transaction has a unique timestamp and each version has a timestamp of the transaction that creates it. So, each transaction tries to read the version that does not cause conflict [24][19].

In addition, the concept of invisible reads allows each transaction to read the value of an object and ensure that the object is not updated until transactions that access it commit [31][32]. However, some STMs use visible read, where a transaction adds itself to the object's reader list, and then it reads the object [24][31][32].The read-only transactions commit directly but update transactions have to check the reader list of each object in its access set to detect conflicts. This way the update transaction does not update any version if that invalidates other reads and causes conflict.

The correctness of the execution can be achieved by satisfying opacity consistency property [15]. Opacity means that all transactions that are executed concurrently must appear as if they are executed in sequential order. Opacity also considers the aborted transactions. Most transactional memory systems use opac-

ity to ensure correctness [16]. Opacity helps to avoid conflicts by preventing some writes. It prevents to write to any object if it is being read by a live transaction. Indeed, opacity prevents those writes through aborts which is very expensive [16][15]. To reduce the cost of aborts, we try to reduce the number of aborts by relaxing the opacity property.

In fact, the idea of relaxing the opacity has also been introduced by Siek and Wojciechowski [35] in a different way than ours. They proposed a relaxed model that allows for update transaction A to update any object that belongs to the read set of a live transaction B, if the live transaction B would not access that object any more [35].

Another approach proposes an approximate consistency [36]. The strategy classifies data in the database as sensitive or non-sensitive. Sensitive data requires strong consistency and full precision, while non-sensitive just requires approximate solutions. They show that in real life there are some systems that do not require returning results with high precision but the approximated results are acceptable. Actually the approximation is acceptable up to some degree which is decided based on the system and data sensitivity [36].

Yu and Vahdat [38] propose an approach for database systems that balances between the availability of replications and the consistency. According to a system needs, they decide to increase the availability by placing more copies of the objects into caches and local memories. However, this way affects the consistency level. Otherwise, they raise the consistency by reducing the copies of the objects.

However, we use an approximate consistency (approximated opacity) on TM up to a degree K. In K-opacity we allow to use one of the most recent K versions of the objects which will be explained in detail in the following chapters.

Also, Herlihy [20] defines the transaction to be a piece of code that executes the operations atomically. We use the same concept with our K-opacity to execute counting (mathematical) operations and queue operations in addition to the regular reads and writes.

For garbage collection, 4-versioned and 8-versioned [27][29] STMs, keep a fixed number of versions for each object. However, fixed number of versions may not be accurate, as for example, is the 4-versioned STM, we may have a situation where we need more than 4 versions for some objects to maintain consistency without aborting transactions.

STM Selective Multi-versioning (SMV) avoids the use of a fixed number of versions and keeps old object versions that may still be useful to some readers [27]. SMV tries to distinguish among objects by the number of accesses. Consequently, for infrequently-updated objects most of the time, it keeps only one version while it keeps more for the others. In our garbage collecting procedure we try to reduce the number of versions, even for frequently-updated objects by reducing the version creation rate and fulfilling the needs of live transactions.

Chapter 3
Notation and Definition

TM consists of several shared memory objects that can be accessed by multiple transactions which execute different kinds of operations [20]. The system contains many threads that execute possibly multiple transactions, where each transaction is a piece of code that accesses multiple shared objects in TM [20]. Each transaction contains a sequence of operations and, in addition, has operations to local objects.

3.1 Kinds of Operations and Sequential Object

In this research, transactions may perform *read/write operations*, *counting operations* such as addition and subtraction, and *queue operations* such as enqueue and dequeue.

i. Read/Write Objects

   Let x be a shared read/write object. The object supports the *read operation* $x.r()$ which returns a stored data in the object, and it also supports the *write operation* $x.w(data)$ which writes the value $data$ to x. A sequence of operations on object x is *legal* if every read operation on x returns the most recent written value to x (or the initial value if there is no previous write operation). We can relax this definition, and we say that a sequence of operations to object x is *K-legal* if every read operation returns one of the K most recent written values to x (or also the initial value if the number of previous write operations is less than K). Clearly, for $K = 1$, a 1-legal execution is also a legal execution. In the example below the sequential execution is 1-legal for object x, and 2-legal for object y, since the last $y.r()$ operation returns the value written by the second to the last write operation to y. (Returned

values are shown below respective operations.)

```
x.w(1) y.w(1) x.w(2) y.w(2) x.r() y.r()
                            2     1
```

ii. Count Objects

Let x be a shared count object. The object supports the *add operation* $x.add(data)$, which returns the value of x and adds *data* to the current value of x. A sequence of operations on object x is *legal* if every add operation returns the last value written to the object by the immediately previous add operation (or the initial value if there is no previous add operation). We relax this definition, and we say that a sequence of operations to object x is K*-legal* if every add operation returns the value written by one of the K most recent add operations applied to the object (or also the initial value if the number of previous add operations is less than K). In the example below assume that count objects x and y are initialized to value 0. The sequential execution is 1-legal for object x, and 2-legal for object y, since the last add operation $y.add(3)$ returns the outcome of the second to the last add operation to y. (Returned values are shown below respective operations.)

```
x.add(1) y.add(1) x.add(2) y.add(2) x.add(3) y.add(3)
0        0        1        1        3        1
```

iii. Queue Objects

Let x be a shared queue object. The object supports the *enqueue operation* $x.enq(data)$ which inserts an element *data* in the queue, and it also supports the *dequeue operation* $x.deq()$ which removes an element from the queue and

returns it. A sequence of operations on object x is *legal* if every dequeue operation returns the oldest inserted element to the queue (if the queue is empty then it returns *nil*). We relax this definition, and we say that a sequence of operations to object x is *K-legal* if every dequeue operation returns one of the K oldest inserted elements to the queue (or also nil if the number of elements in the queue is less than K). In the example below assume that queue objects x and y are initially empty (hold no elements). The sequential execution is 1-legal for object x, and 2-legal for object y, since the last dequeue operation $y.deq()$ returns the second to the oldest inserted element to queue y. (Returned values are shown below respective operations.)

```
x.enq(a) y.enq(c) x.enq(b) y.enq(d) x.deq() y.deq()
                                    a       d
```

3.2 Approximated Opacity

When a transaction executes all of its operations, the transaction tries to commit by executing function TryC(), which validates the object values in the concurrent execution and based on that decides to commit or abort. Commit means that all changes made to the shared objects in the transaction are saved to shared memory, but abort means to ignore the changes. In fact, each transaction has a *status* that initially equals to *live* and updated to *committed* or *aborted* after transaction validation. Also, *read-only transactions* have only read operations (to the shared objects), while *update transactions* have at least one write operation (to the shared objects).

In our model, when a transaction arrives it gets a unique timestamp i [29][24]. We will use the timestamp i as an identifier, as for example, T_i means a transaction with timestamp i, and T_i arrives before T_j if $i < j$.

A transaction T_i has a *read set* $T_i.rSet$ (*write set* $T_i.wSet$) which contains all shared objects that would be read (written) by T_i. Note that an object may belong in both the read and write sets. A transaction's access set $T.accSet$ consists of all elements in both read and write sets, namely, $T.accSet = T.rSet \cup T.wSet$.

A transaction execution consists of a sequence of operations' executions. Every operation execution is represented with two events which are the operation invocation (when operation execution starts) and respective response (when operation execution ends). Every event is instantaneous and it has a real time that it occurs. The event is used to indicate different situations of the system. The time between the invocation and response is the operation interval. In addition, in the execution of a transaction there are three special intervals, one for obtaining the timestamp and the second for the function call TryC(), and the third is to commit by calling Commit() or abort by calling Abort(). The timestamp interval is the first interval in the transaction, while the TryC() and Commit() or Abort() intervals are the last. Based on the response of TryC(), the *status* of the transaction changes from *live* to either *committed* or *aborted*. None of the intervals within the transaction overlap with each other. The interval of the transaction is the time period between the first invocation event and the last response event.

A pending operation (of a function call) is one that starts but has not yet finished, or in other words, there is an invocation event but no respective response event. A pending (live) transaction is the one that either contains a pending operation, or it doesn't have an invocation event for at least one of its operations or the TryC(), Commit() or Abort() function calls.

A history H is a sequence that includes all events of all involved transactions [15]. With respect to history H, we can use the relation order $<_H$ to define *the real time order* of the transactions, such that for each two transactions T_i and T_j in H,

$T_i <_H T_j$ if all events of T_i happen before all events of T_j. We say that the relation $<_H$ *respects the real time order* of the transactions if $T_i <_H T_j$ implies $i < j$ (recall that i and j are timestamps in our notation). From now on we will only consider histories that respect the real time order of transactions. The relation $<_H$ may be a partial order on the transactions, since if for some pair of transactions T_i and T_j in H, it could be that the first event invocation of T_j happens before the last response event of T_i; namely, there is an execution overlap between T_i and T_j [20][24][15].

The complete history of H, denoted as *complete*(H), is obtained by defining the *status* for each pending transaction. If a pending transaction didn't invoke TryC(), its *status* is *aborted*, while in any other case the status can be either *committed* or *aborted* [15]. Since History *complete*(H) may include aborted transactions, we have to define the legality and K-legality carefully to discard the effects of all operations that belong to the aborted ones. Based on what we mentioned before, we extend the definitions of the legal and K-legal operations to match with the transactional manner.

A history S is *sequential* if it is complete, and all transactions in S appear as if they execute sequentially, namely $<_S$ is a total order. In other words, in S there are no pending transactions, and moreover, every pair of transactions is ordered, so that the intervals of any two transactions do not overlap. Suppose that S is equivalent to H; they have the same set of events. Also, we say that S preserves the *real time order* of H if for any two transactions T_i and T_j, $T_i <_H T_j$ implies $T_i <_S T_j$ [20][24].

Now, to define the legality to the respect of transactions we need to be very careful since we have to discard the operations that belong to aborted update transactions. Therefore, let S be a sequential history. Now, for any transaction T_i denote by S_i the subsequence of S that includes T_i and all committed transactions

that appear in S before T_i. In other words, for any transaction $T_j \in S_i$, either $j = i$ or T_j committed and $T_j <_s T_i$.

In addition, for any transaction T_i there is V_i which is a set of all the write, counting and queue operations in S_i; in other words, V_i contains all operations that are visible to T_i. However, with respect to any shared object x, we can define $V_i(x)$, as a subset of V_i that considers only the operations on x.

Based on the above, a *legal operation* by T_i on object x is the one that reads the value of the last operation in $V_i(x)$. Moreover, a *K-legal operation* on the object x is the one that reads the value of one of the K last operations in $V_i(x)$. For this legality definition, we assume that for a read/write object, there is at most one write operation for each transaction, which is the last write operation for the object within the transaction. Similarly, for a count object, there is at most one representative add operation in a transaction, which aggregates all the individual add operation arguments in the transaction. On the other hand, for a queue object each individual enqueue or dequeue operation is considered in the legality specification of the object.

Thus, T_i is *legal*, if all operations in T_i are *legal*. Then, S is *legal* if all transactions in S are *legal*. Also T_i is K-legal, if all operations in T_i are K-legal. Then, S is K-legal if all transactions in S are K-legal. Consequently, a legal history is a special case of K-legal history by taking $K = 1$.

In FIGURE 3.1 we give an example that demonstrates the notion of legal and K-legal executions on a read/write object x. In FIGURE 3.1(a) transaction T_1, T_2 and T_3 arrive consecutively. Transaction T_2 has a read operation that returns the value 1 which is a legal read because it reads the initial value of x. Then, T_2 writes the value 2 to x and it commits. Transaction T_1 has illegal read as for some reasons, it returns 1 while the last write to x writes the value 2, so it aborts. However, T_3

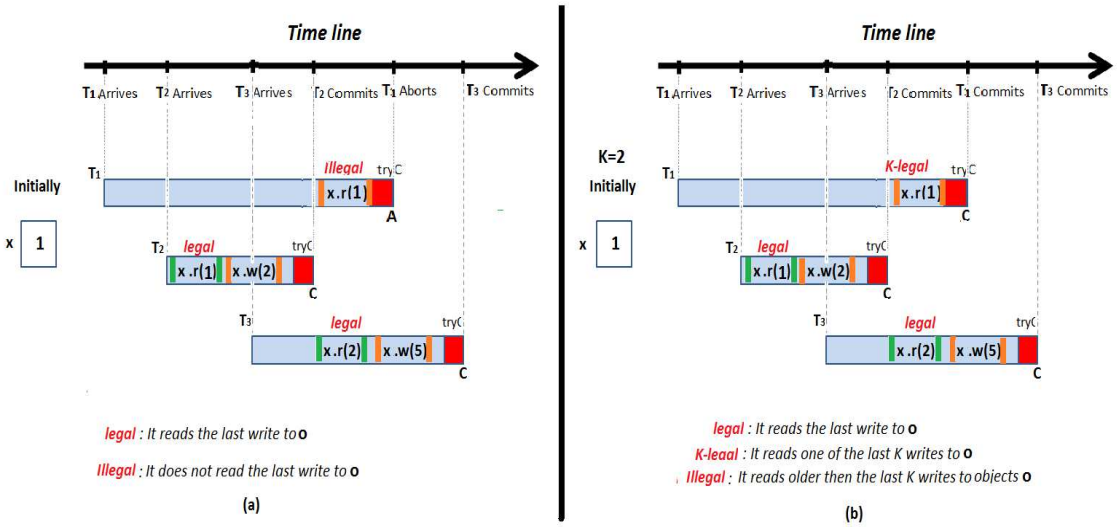

FIGURE 3.1. (a) Example of Legal Reads on Read/write Object, (b) Example of K-Legal Reads on Read/write Object

reads 2 which is the last written value to x and that is legal. It also writes to x and commits.

In FIGURE 3.1(b) we show how K-legal allows to commit more transactions. Suppose $K = 2$, which means it is allowed to read one of the last two writes on x. Now, the three transactions arrive. Transaction T_2 reads the initial value of x which is 1, so it is a legal read. Then, it writes 2 to x and commits. Thus, the last two writes on x are 1 and 2 (we consider the initial value of the object as the first write). Transaction T_1 has a read operation that reads the value 1 which is the second last write to x. As $K = 2$ it is K-legal, and then T_1 commits. Moreover, the read operation in T_3 reads the last write to x which is legal. After that, it writes to x and commits.

We say that two histories are *equivalent*, if they have the same set of events.

Definition 3.1 (Opacity)**.** *A history H is opaque if it has an equivalent complete history H' which further has an equivalent sequential history S such that:*

- *H' preserves the real time order of transactions in H, namely, $<_H \subseteq <_{H'}$.*

- *S preserves the real time order of transactions in H′, namely,* $<_{H'} \subseteq <_S$.
- *S is legal with respect to all of the involved operations.*

Definition 3.2 (K-approximated opacity)**.** *A history H is K-opaque if it has an equivalent complete history H′ which further has an equivalent sequential history S such that:*

- *H′ preserves the real time order of transactions in H, namely,* $<_H \subseteq <_{H'}$.
- *S preserves the real time order of transactions in H′, namely,* $<_{H'} \subseteq <_S$.
- *S is K-legal with respect to all of the involved operations.*

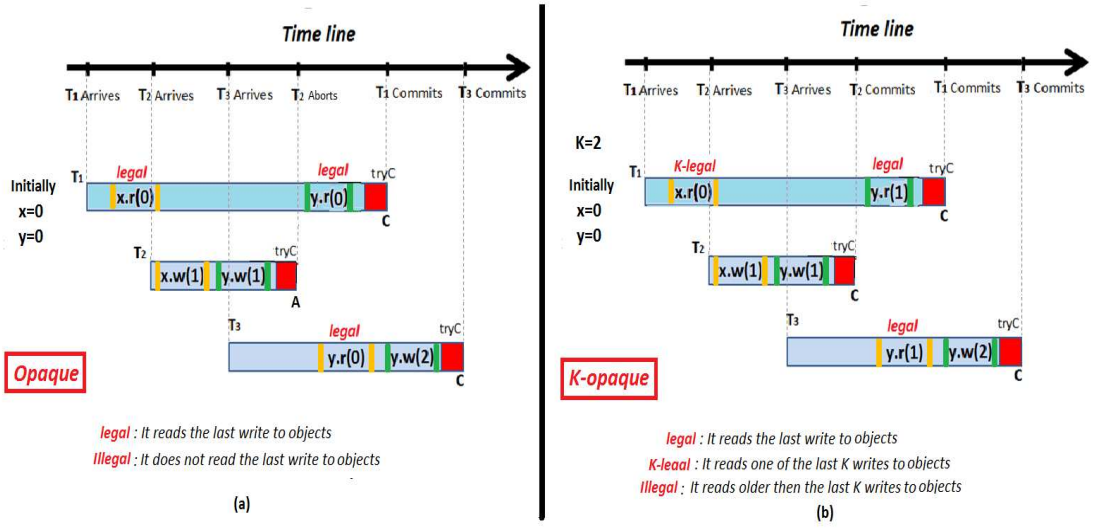

FIGURE 3.2. (a) Example of Opaque Execution for Transactions that Access Read/write Objects, (b) Example of K-opaque Execution for Transactions that Access Read/write Objects

FIGURE 3.2 shows two executions of three transactions T_1, T_2 and T_3, sharing two read/write objects which are x and y. The respective histories H_a and H_b are shown below. It is clear that the histories H_a and H_b of the two executions are complete as there is no live transaction or pending operation, and they preserve real-time order. FIGURE 3.2(a) shows an opaque execution (H_a) that has to abort

T_2 as it violates the validity of T_1. By aborting T_2, there is an equivalent legal sequential history S, such that $S = \langle T_2 <_s T_1 <_s T_3 \rangle$. So H_a is opaque.

FIGURE 3.2(b) illustrates H_b which is not opaque. This is because if T_2 commits then T_1 cannot be ordered before T_2 as T_1 reads $y = 1$, and it cannot be ordered after T_2 as it reads the initial value of x, which means it reads before T_2 writes. However, H_b is K-opaque (where $K > 1$), as there is an equivalent K-legal sequential history S, such that $S = \langle T_2 <_s T_1 <_s T_3 \rangle$. For $K = 2$, T_1 reads one of the last K writes. Furthermore, the first read operation in T_1 returns $x = 0$, which is the second last write to x (considering the initial value as a write). So, H_b is K-opaque.

$$H_a = \langle x_1.r(0), x_2.w(1), y_2.w(1), TryC()_2, Abortted_2, y_1.r(0), y_3.r(0), TryC()_1, Committed_1, y_3.w(2), TryC()_3, Committed_3 \rangle$$

$$H_b = \langle x_1.r(0), x_2.w(1), y_2.w(1), TryC()_2, Committed_2, y_1.r(1), y_3.r(0), TryC()_1, Committed_1, y_3.w(2), TryC()_3, Committed_3 \rangle$$

Definition 3.3. *The approximately opaque permissive multi-version transactional memory never aborts read-only transaction while the read operations in the read-only transactions may return approximated results, and update transaction is aborted only if it conflicts with another transaction.*

Chapter 4

Transactional Memory Scheduling Using Machine Learning Techniques

4.1 Introduction

This chapter is previously published by IEEE [1]. Clearly from one system to another, there are different kinds of data structures, hardware capabilities and transactions' types. Also transactions have different kinds of operations and durations. Thus it is not easy to propose an algorithm or a technique that works well with all systems and under any conditions. Therefore the main focus of this chapter is to propose a flexible scheduling algorithm that is able to adjust itself to perform well under different conditions. To achieve such flexibility, we support our scheduling algorithm with a machine learning technique that checks the history of transactions in the system and based on that we adjust the scheduling process. We apply our algorithm on a famous and classic algorithm (LSA) to show the benefits of our algorithm. Actually, in this chapter we just focus on the read/write operations on the shared read/write objects. In this chapter, we illustrate the design of the supervised machine learning models and the design of the unsupervised machine learning model. Also, we discuss the LSA algorithm and our scheduling algorithm. After that, we present the experiments and some results that show the effect of learning on the performance of LSA.

4.2 The Supervised Machine Learning Techniques and Dataset

To use a supervised model we have to generate a *dataset* that includes some training examples. The training examples show how to map the inputs (features) to

[1] This chapter previously appeared as [Basem Assiri and Costas Busch, Transactional Memory Scheduling Using Machine Learning Techniques, published by The Institute of Electrical and Electronics Engineers (IEEE)]. See the letter in Appendix.

the suitable FV. In our *dataset*, there are three features, which are the number of reads-only, the number of updates and the order of the transactions which we call the sequence length. In fact, we track the prefix of transactions' history and pass it to the learning model. We count how many read-only transactions and how many updates are in the prefix. Also, we convert the order of reads and updates into one number and we call it the sequence length.

TABLE 4.1. Two Different Scenarios of 10 Transactions and How to Calculate the Sequence Length (r means a read-only transaction and u means an update transaction)

Scenario 1	The sequence	The Sequence Length	Scenario 2	The Sequence	The Sequence Length
$r1$	5, 5	2	$u1$	1, 1, 1, 1, 1, 1,	10
$r2$			$r1$	1, 1, 1, 1	
$r3$			$u2$		
$r4$			$r2$		
$r5$			$u3$		
$u1$			$r3$		
$u2$			$u4$		
$u3$			$r4$		
$u4$			$u5$		
$u5$			$r5$		

TABLE 4.1 shows an example of how to calculate sequence length for only 10 transactions with different orders. In the first scenario, there are 5 reads followed by 5 updates. The sequence consists of two numbers which are 5 and 5, so the sequence length equals to 2. In the second scenario, the sequence consists of 1 update, 1 read, 1 update, 1 read and so on. It consists of 10 numbers, so the sequence length is 10.

Our *dataset* consists of four columns which are the three features followed by the suitable FV. To generate the *dataset* we first need to know the size of the prefix which is a piece of history we track to extract the pattern from. Then we need to find all possible combinations of the three features. For example, if the prefix size is 20, the first combination is 1 read-only, 19 updates and the sequence length is 2, which means the read-only transaction is the first or the last one in the prefix. The

next combination will be 1 read-only and 19 updates and the sequence length is 3, which means the read-only transaction is somewhere in the middle of the prefix, and so on. Then we pass each permutation as an input set to the LSA and we run it with all FVs we want to test. During the runs we record the throughput of all FVs on our algorithm, and the FV with the maximum throughput will be the suitable class. This way we generate all training examples in the *dataset*.

For simplicity and to avoid learning process overhead, in our experiment we reduce the size of the prefix to 10 transactions. In our experiment, we test all FVs from 1 to 9, and according to the results we select only three FVs which are $1, 4$ and 8. The FV$= 1$ means 1 read-only transaction per 9 updates. The FV$= 4$ means 4 read-only transactions per 6 updates, while FV$= 8$ means 8 read-only transactions per 2 updates. More details about FV selecting will be illustrated later in section 4.5.

TABLE 4.2. Some Training Example from our Dataset

54	4	1	4	8
1	9	2	1	
2	8	2	1	
2	8	5	1	
:				
:				
5	5	2	8	
5	5	5	4	
5	5	10	8	
:				
:				
9	1	2	8	
:				

TABLE 4.2 shows a sample of our *dataset* which consists of rows and columns. The first row gives a summary of *dataset*. It states that the number of examples is 54, the number of columns is 4, and shows the three classes. Then, each example is placed individually in a row. For example, in the row number 2 in the table, the

example 1, 9, 2, 1 means that if there are 1 read-only, 9 updates and the sequence length is 2, then the suitable FV is 1.

4.2.1 Support Vector Machine SVM

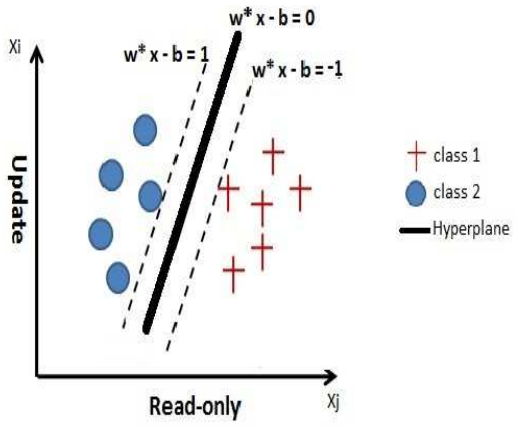

FIGURE 4.1. SVM

The basic SVM takes a set of input data and predicts the suitable output, so each given input will be classified into a suitable class [23]. FIGURE 4.1, shows the SVM classification accuracy which is calculated according to the following formula:

$$w * x + b = 0, where$$

x denotes the features

w the normal vector to the hyper plane

b denotes misclassification

Higher accuracy of the classification using SVM requires bigger margins among classes.

At the beginning we need to test the classification accuracy of SVM and KNN. We design a *dataset* with only two features, which are the number of read-only

and update transactions, and we have only 3 FVs. We use Scikit-learn to test the machine learning models in this thesis (Scikit-learn is a package that is designed to introduce many machine learning algorithms and codes (in Python) in a simple and understandable way) [26]. The result of running the SVM classifier using the *dataset* of two features is shown in FIGURE 4.2 (a). In FIGURE 4.2 (a), the x-axis represents the number of read-only and y-axis represents updates. The classes represent the three FVs. Each point in the figure represents a training example. The accuracy of this classification was about 73% which is not high. Some dots are classified in the wrong class. It is clear that there is misclassification, which decreases the classification accuracy. Therefore, the SVM fails at some point because SVM accuracy increases when the margin between classes is bigger. However, in our model we need to classify some dots where the number of read-only transactions is close to the number of updates and those usually affect the accuracy of SVM [23].

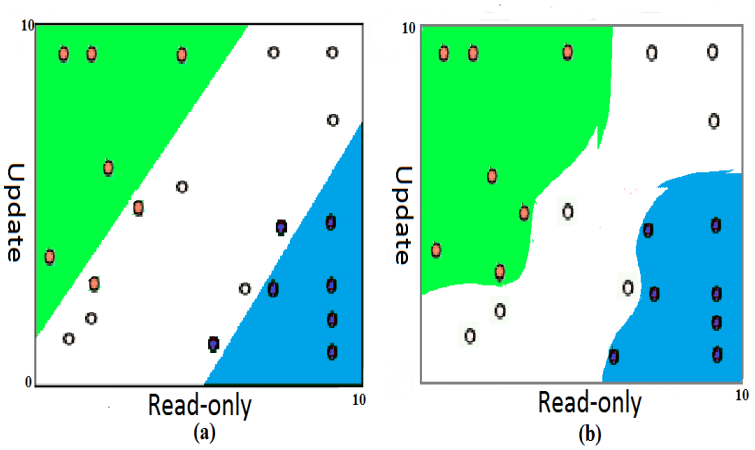

FIGURE 4.2. SVM and KNN with the Dataset of Two Features

4.2.2 K-Nearest Neighbor

The K-Nearest Neighbor algorithm (KNN) is a classifier that classifies objects based on closest training examples in the feature space [3][33]. FIGURE 4.2 (b)

shows the implementation of KNN on the *datasets* of two features [26]. FIGURE 4.2 (b) shows that all dots are classified under the correct class. The accuracy of KNN is higher that SVM as it achieves about 90%. Apparently, the KNN model is nonlinear; it is able to classify the critical dots. In short KNN model is more suitable for our system than SVM. Thus, we decide to use KNN to support the transactions' scheduling process.

However, as aforementioned we have to run the classifier on the *dataset* of the three features that appears in TABLE 4.2. FIGURE 4.3 shows the result of the KNN classification on the three features where each feature is represented by an axis on the graph. The KNN classifies the data example into three classes which are red, blue and yellow representing the FVs $1, 4$ and 8 respectivly.

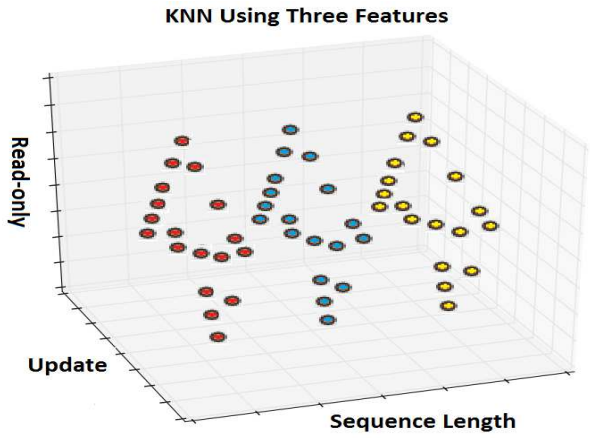

FIGURE 4.3. KNN with our Dataset

4.3 The Unsupervised Machine Learning Model

HMM is one of the unsupervised machine learning models. In unsupervised machine learning techniques the data appears without labels. So, in HMM there are unobserved hidden states X which must be discovered based on the observed data Y. HMM consists of some states and all states have probabilities that help to predict the hidden state [21]. For example, it is difficult to find a scheduling model that

suits all systems by increasing throughput and reducing conflicts. Therefore, the suitable transactions' scheduling for each system is the hidden state. However, we can find the hidden state based on some observations such as system performance, the transactions' history and transactions duration. Furthermore, by recording observations and states frequently, the first result helps to predict the second one and so on. Thus, HMM calculations rely on the following:

i. The initial probabilities for hidden states.

ii. Transition probabilities that tells how to transit among states over time.

iii Observation probabilities which are used as indications to find out the hidden state.

In our model the hidden states are the suitable FVs. As we mentioned above the FVs we use are 8, 4 and 1. FIGURE 4.4 shows the initial probabilities which we need to start. The probabilities of FVs 8 and 1 are .33 while it is .34 for FV 4, so the system starts with state FV= 4. Even if we start with inaccurate probabilities, the model can improve itself over time by maintaining probabilities based on the observations. Also, as the probability of X changes over time, we propose transition probabilities, and we favor remaining in the current state. Therefore, the state of X at first unit of time T_1 is more likely to remain the same at the second unit of time T_2. The probability of switching from $X = 8$ or from 1 to 4 is .35 and it is less likely to transit between 8 and 1 directly since the probability is just .15. The probability of switching from $X = 4$ to 1 or to 4 is .25. The two states at the bottom of FIGURE 4.4 shows the probabilities of our observed data, which is $Y1$ = read-only and $Y2$ = update. Thus, the more reads favors the FV= 8, the more updates favors the FV= 1 and the equal number of both suits the FV= 4.

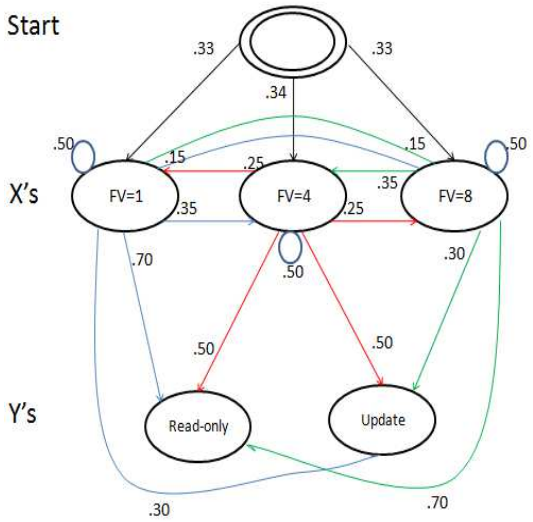

FIGURE 4.4. HMM and the Probabilities (Start is the initial state, X's are the states and Y's are the observations)

FIGURE 4.5 shows how to build the sequence of states over time. It means to find the state of X_4, we need to calculate the probability of initial state X_1, the transition probability from X_1 to X_2, the probabilities of Y_1 using X_2, the transition probability from X_2 to X_3, finding Y_2 using X_3, and the transition probability from X_3 to X_4. Indeed, we get the probability from the model (in FIGURE 4.5) and we multiply them to calculate X_4.

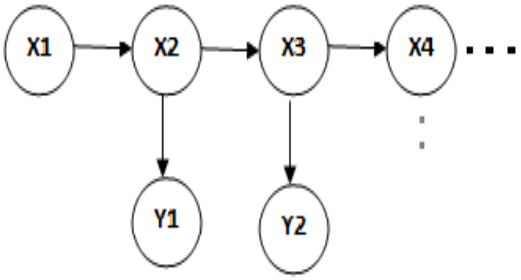

FIGURE 4.5. Sequence of States in HMM

4.4 The Design of our Algorithm

4.4.1 The Design of LSA

First, we need to talk about LSA [29]. Using LSA, each object in the memory may have more than one version. This algorithm is used to execute transactions in paral-

lel where some read and write operations may interleave and affect the correctness of each other. The correct concurrent execution can be verified by extending the execution to match a sequential valid execution; otherwise, it is incorrect. Many algorithms run transactions and validate the transaction's execution at the end to commit or abort, but LSA verifies the consistency at each object access point of time. Thus, an LSA is able to verify the consistency of execution during each object access by building consistent snapshots. This can accumulatively verify the validity and the correctness of the execution.

When a transaction starts execution, it sets the lower bound of its snapshots. The read operations try to read the last written version of the object. However, if the last written version was created after the transaction starts, it checks if this version violates the consistency and, if it does, it ignores that version and reads an older one which has validity range that suits the transaction. This way LSA selects the version that produces a consistent snapshot. The write operations are executed locally and they create new versions at transaction's commit time.

Consequently, read-only transactions commit directly and they do not abort. The update transaction must validate the versions to commit or abort, and if it commits it has to get a unique commit time [29].

4.4.2 Scheduling Algorithm

At the beginning of the execution, our algorithm executes transactions using LSA. We suppose that we know whether the transaction is read-only or update at the beginning of the execution. (In many systems, the kinds of transactions are identified at the beginning such as read balance and bank statements in bank systems, or products quantities in inventory systems). We record the transactions' numbers, order and types in $recordingArray[]$. The size of the $recordingArray[]$ is related to how much data we want to investigate. It is preferred to keep it as small as possible

to avoid the learning overhead and its negative impact on the performance. Then, we pass the recorded data to the learning model which is either KNN or HMM and we conduct online and real-time learning. The learning model processes the data and returns the best FV that suits the execution. After that, the model uses the proposed FV to schedule transactions on LSA.

In fact, we need to have continuous learning however that influences the performance of the system. Therefore, we add flexibility to the algorithm using $frequent$, which is a number used to decide how frequently we call the learning model. The stability and consistency of the numbers, types and arrival times of transactions differ from one system to another. So, if the system is stable then we have a large value for $frequent$, meaning we call the learning model infrequently.

In our algorithm, each transaction arrives increase a global counter which is called $transactionCounter$. The transaction first checks the $frequent$ to know if it is time to call the learning model or not. Then, if it is the time to call the learning model, it starts recording transactions. In the recording process, each transaction is represented in the $recordingArray[]$ as a 0 if it is a read-only and as a 1 if it is an update transaction. Then, it executes using LSA. After the, recording finishes, we pass the $recordingArray[]$ to the learning model (KNN or HMM) which returns the FV. The learning model returns the number of read-only $fvReadonly$ and from that, we calculate the number of updates $fvUpdate$. From this point, we schedule transactions based on the FV we have. Indeed, we ignore this FV just in case there is only one type of transaction. In the algorithm, we use $readonly$ and $update$ which are counters telling how many pending transactions there are of each type.

4.5 Experimental Results and Discussion

In our experiment, we use standard benchmarks to verify the benefit of learning on transactional memory scheduling. In fact, we use Linked-list, Red-black Tree

Algorithm 1: The Algorithm

```
/* Global variable initialization */
transactionCounter ← −1;
size ← x; //To set the size of recordingArray[]
recordingArray[size]; //It records the transactions order and type
recording ← false;
frequent ← number. //It tells how frequently we call learning model.
r ← 0; //Counts reads
u ← 0; //Counts updates
fvReadonly; //The value assigned by learning model
fvUpdate; //The value assigned by learning model
i ← 0;
readonly ← 0; //Counts the pending reads-only
update ← 0; //Counts the pending updates

Upon receipt of a transaction do;

//We use an atomic operation to increase transactionCounter
transactionCounter.getAndInc();
//Check if it is the time to record transactions and call learning model
if ((transactionCounter  mod frequent) = 0) then
    recording ← true;
if (recording = true) then
    i++;
    if (i < size) then
        if (TransctionType is read-only) then
            //0 means read-only
            recordingArray[i] ← 0;
        else
            //1 means update
            recordingArray[i] ← 1;
    else
        //The learning model is called and returns the FV
        fvReadonly ← LearningModel(recordingArray[i]);
        fvUpdate ← (10 − fvReadonly);
        i ← 0;
        //To stop recording
        recording ← false;
//Start execution using LSA
LSA(transaction);
```

and Bank benchmarks from TinySTM-1.0.5 [13]. We run the experiments on a machine with dual Intel(R) Xeon(R) CPU E5-2630 (6 cores total) clocked at 2.30 GHz. Each run of the benchmark takes about 10000 milliseconds using 10 threads. In the Bank benchmark, there are three kinds of operations which are read balance, write amount and transfer from one account to another. Read balance is a read-only transaction, while write amount and transfer are update transactions; we consider any transaction that includes at least one write operation as an update. Initially, there are only 20% reads-only and 80% updates. The benchmark executes millions of transactions that access 1024 bank accounts in parallel.

```
CONTINUE...
else
    if ( TransctionType is read-only) then
        readonly.getAndInc();
        //When it exceeds the FV and there is no update transactions
        while (r > fvReadonly) ∧ (update = 0) ) do
            wait();
        r + +;
        LSA(transaction); //Start execution using LSA
        readonly.getAndDec();
    else
        update.getAndInc();
        //When it exceeds the FV and there is no read-only transactions
        while ((u > fvUpdate) ∧ (readonly = 0)) do
            wait();
        u + +;
        LSA(transaction); //Start execution using LSA
        update.getAndDec();
    if ((r > fvReadonly) ∧ (u > fvUpdate)) then
        r ← 0;
        u ← 0;
return;
```

In Linked-list and Red-black Tree benchmarks, there are three kinds of operations which are add node, delete node and contain which searches for specific value in the list or tree. Contain operations are considered as a read-only but add node and delete node operations are update transactions. In both, initially there are 80% reads-only and only 20% updates.

However, we test the three benchmarks with three different scenarios which are 80% reads-only and 20% updates, 50% reads-only and 50% updates, and 20% reads-only and 80% updates.

First, we run each benchmark against all FVs to obtain a general idea about FVs themselves and how they affect the performance. The FVs we use are ratios out of 10 such that the FV= 9 means 9 : 1, where the first number represents the number of reads-only and the other represents the number of updates. We test FVs from 1 to 9 with all three benchmarks and all scenarios.

FIGURE 4.6 (a) demonstrates the throughput of FVs with the Linked-list benchmark using different percentages of reads and writes. The figure shows that the performance improves as we increase the number of read-only transactions and that happens with all three scenarios. It is clear that the scenario of 80% of reads only is the best, while throughput drops as we increase the number of updates. Also, within each scenario, large FVs allow more reads-only to be scheduled concurrently which increases throughput as well. This happens because the reads-only usually have less durations than updates and do not abort as they do not affect the consistency. This increases the throughput and improves the performance.

The same thing happens with the Red-black Tree benchmark, FIGURE 4.6 (b) shows the Red-black Tree throughput with all FVs. The figure shows dramatic increment in the throughput for the three scenarios when the FV allows more reads-only (using large FVs) because the reads-only is much faster than updates.

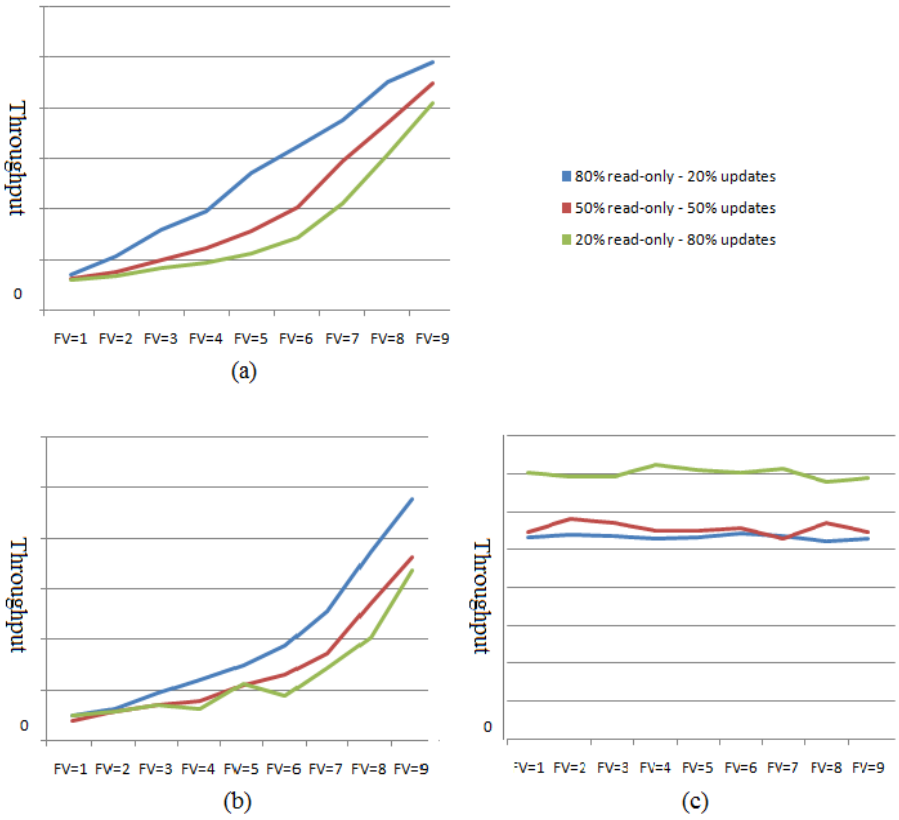

FIGURE 4.6. (a) The Linked-list Throughput (Commits Per Time) with All FVs. (b) The Red-black Tree Throughput with All FVs. (c) The Bank Throughput with All FVs.

In updates, transaction has to traverse the entire tree to the leaf and it may need to change the colors of some nodes and the shape of the tree which make the update transaction much longer. Thus, raising the number of reads-only through the percentage of generated reads-only or through scheduling improves the performance.

FIGURE 4.6 (c) illustrates the Bank benchmark where all FVs have almost the same performance with all scenarios. This happens because of the data structure type, and because of the differences between the duration of read-only transactions and update ones is very small. Indeed, the duration of read-only transactions here is a bit longer than the duration of updates. Thus, the scenario of 20% reads-only and 80% updates achieves the top performance.

Furthermore, having all FVs as classes in the learning model will affect the accuracy and increase the learning overhead. Therefore, we need to reduce the number of FVs in the learning model based on the results shown in FIGURE 4.6. We select only three FVs which are $1, 4$ and 8 to be the classes in our learning process. Apparently, FIGURE 4.6 shows that the performance of FV$=1$ is very close to FV$=2$, the FV$=4$ is comparable to $3, 5$ and 6, and the performance of FV$=8$ is akin to FVs 7 and 9.

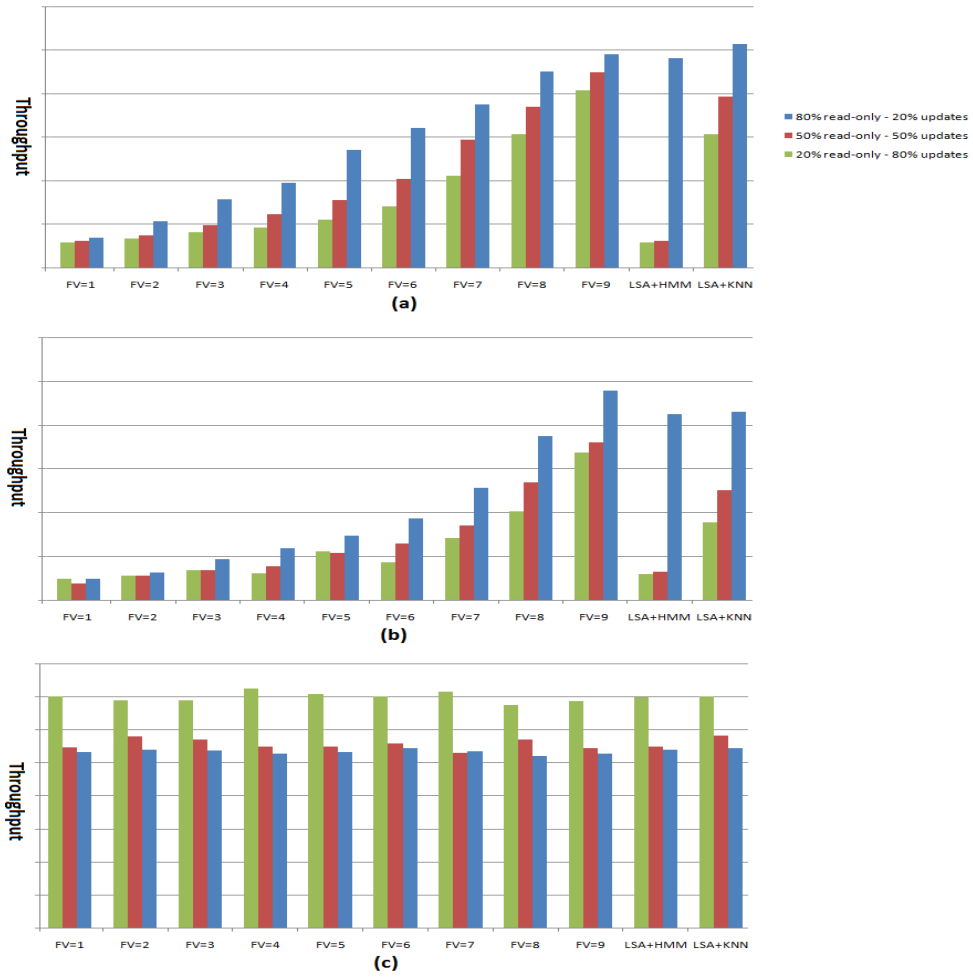

FIGURE 4.7. (a) The Accuracy of LSA Using KNN and LSA Using HMM Compared to the Enforced FVs Using the Linked-lest Benchmark. (b) The Accuracy of LSA Using KNN and LSA Using HMM Compared to the Enforced FVs Using the Red-black Tree Benchmark. (c) The Accuracy of LSA Using KNN and LSA Using HMM Compared to the Enforced FVs Using the Bank Benchmark. (We Measure the Accuracy Based on the Throughput)

In FIGURE 4.7 we show the accuracy of learning models. We run LSA with learning models which are HMM and KNN such that the learning models check the prefix of execution and decide the suitable FVs. Then we enforce different FVs in LSA scheduling and we compare their (FVs) throughputs with throughputs of LSA using HMM and LSA using KNN. Actually we run them on the three benchmarks (Linked-list, Red-black Tree and bank) and we run each one with three scenarios (80% reads-only and 20% updates, 50% reads-only and 50% updates and 20% reads-only and 80% updates).

FIGURE 4.7 (a) shows the throughput of LSA on the Linked-list benchmark where $FV = 9$ shows the best of the enforced FVs. LSA using HMM shows high accuracy in some cases and fails with others. Clearly, HMM returns an accurate FV when there is 80% reads-only and 20% updates but it returns inaccurate FVs with the other scenarios (50% reads-only and 50% updates and 20% reads-only and 80% updates). On the other hand, LSA using KNN returns accurate FVs with all three scenarios.

FIGURE 4.7 (b) shows the throughput of LSA on the Red-black benchmark where $FV = 9$ also shows the best performance (highest throughput) of the enforced FVs. LSA using HMM performs well with the scenario of 80% reads-only and 20% updates but it does not succeed with the other two scenarios. However, LSA using KNN performs well with all three scenarios which indicates the success and accuracy of KNN in the selection of FVs.

FIGURE 4.7 (c) shows the throughput of LSA on the Bank benchmark where all FVs perform almost the same. Thus, the accuracy of HMM and KNN cannot be judged as the selection of any FV does not show any difference on the throughput. However, it is clear that both HMM and KNN do not show negative impact on

the performance because they are designed in a proper way that minimize their overhead.

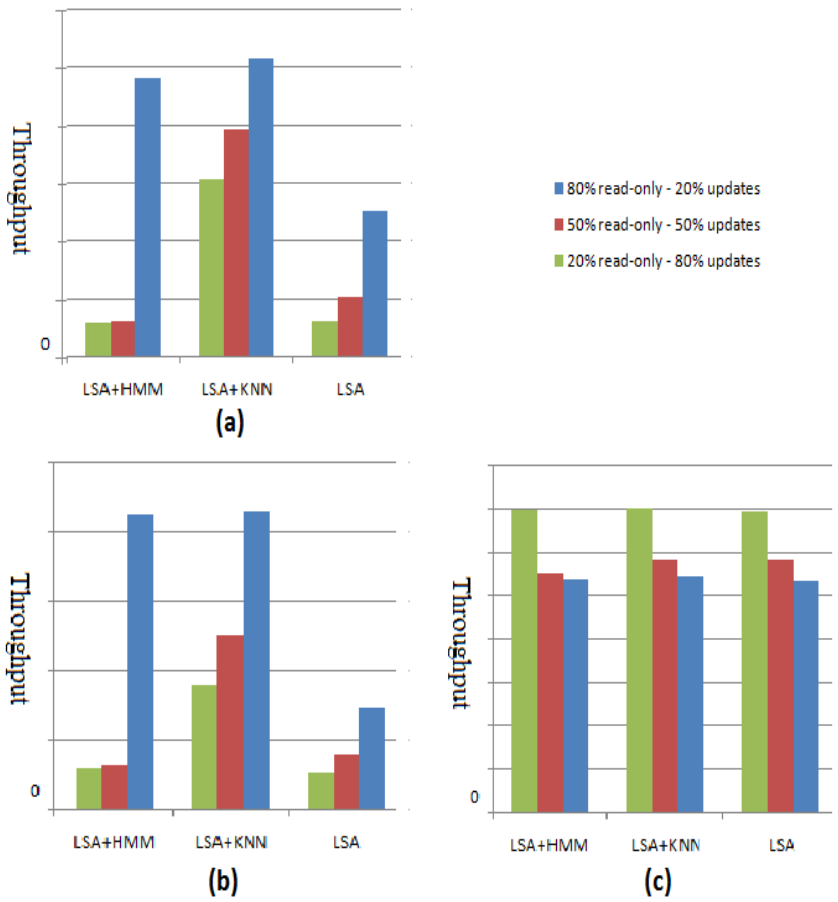

FIGURE 4.8. (a) Comparison of the Throughputs of LSA, LSA with KNN and LSA with HMM Using the Linked-list Benchmark. (b) Comparison of the Throughputs of LSA, LSA with KNN and LSA with HMM Using the Red-black Tree Benchmark. (c) Comparison of the Throughputs of LSA, LSA with KNN and LSA with HMM Using the Bank Benchmark.

FIGURE 4.8 demonstrates a comparison of the performance of LSA using transactions' timestamps scheduling, LSA with KNN and LSA with HMM. Timestamps scheduling means to schedule transactions based on their arrival times. Using the Linked-list benchmark, FIGURE 4.8 (a) shows that high percentage of read-only transactions usually results in high throughput as we mentioned before. FIGURE 4.8 (a) proves the advantage of using machine learning for transactions'

scheduling. For LSA using KNN, it shows the highest throughput with all scenarios. Also, when we use HMM, it improves the performance of LSA only when the percentage of reads-only is high. On the other hand, the performance of LSA with HMM is negatively affected as the percentage of updates increases. The figure also illustrates the risk of the learning process, as the learning accuracy problems lead to unsuitable scheduling which causes more aborts. This clearly happens with HMM when there are 50% or 80% updates. Thus, KNN obviously is more suitable to LSA scheduling.

The same thing happens with the Red-black Tree benchmark in FIGURE 4.8 (b), where KNN is more efficient than HMM and timestamps scheduling. HMM works well only with large number of read-only transactions and it fails when we reduce the reads-only.

In FIGURE 4.8 (c), with the Bank benchmark, the duration of update transaction is less than read-only one. So, the throughput drops when there are more read-only transactions. Furthermore, KNN and HMM do not have significant influence on the performance of the Bank benchmark because the durations of transactions in general are very short, and the durations of updates with aborts almost equals to the durations of reads-only.

Thus, the learning techniques are more helpful if transactions' durations on average are long, and when the costs of updates and aborts are very expensive. Also, the KNN learning model helps to achieve high throughput and better performance of LSA.

Chapter 5

Approximately Opaque Multi-version Transactional Memory for Read-only Transactions

5.1 Introduction

This chapter is previously published by IEEE [1]. In multi-version transactional memory, read-only transactions do not abort, while transactions that update objects may abort. Since any new update creates new version and we still save previous ones, read-only transactions avoid aborting by finding the version that preserves the correctness and satisfies the consistency. In this chapter we improve the space complexity by reducing the number of saved versions. However, some transactions would not be able to find the suitable version which affects the consistency. To cope with this issue, we propose to relax the consistency for some transactions such that they are allowed to read some stale values of the memory up to some limit K. Thus, we use an approximate consistency in transactional memory. We define K-opacity as a relaxed consistency property where transactions' read operations may return one of K most recent written values. In this chapter we apply K opacity only on read-only transactions while the update transactions access only the latest version of an object. Actually, in this chapter we just focus on the regular read/write operations on the shared read/write objects. This chapter shows the design of the K-opaque algorithm and the proof of correctness. After that, we present some experimental results to illustrate the impacts of consistency relaxation on performance.

[1] This chapter previously appeared as [Basem Assiri and Costas Busch, Approximately Opaque Multi-version Permissive Transactional Memory, published by The Institute of Electrical and Electronics Engineers (IEEE)]. See the permission letter in Appendix.

5.2 Design of the Algorithm

Our multi-version algorithm (Algorithm 2) is a timestamp-based algorithm (similar to previous works that do not consider approximated opacity proposed [24][12]). As shown in FIGURE 5.1, each shared object x has multiple versions v that are stored in list $x.vl = (v_1, v_2, ..., v_n)$. A version is denoted as $v_i = (ts, data, rl)$, where ts represents the timestamp of the transaction that writes this version, $data$ is the value of x, and rl is a reader list that includes the timestamps of all transactions that have been reading this version. In our algorithm, we create a new version of x each K commits and we save it in version list $x.vl$. The last written value is maintained in $x.lastCommit = (ts, data, rl)$ which is overwritten with each commit on x to record the last write.

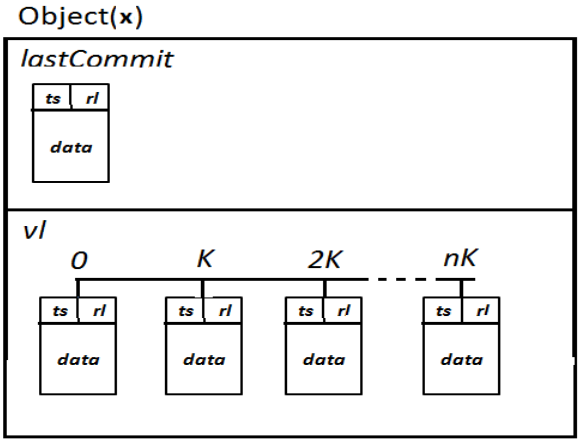

FIGURE 5.1. Object Data Structure

In our algorithm, there is a special *timestamp* object with a respective atomic *getAndIncrement()* method which increments the *timestamp* value by 1. So, each transaction that arrives is given a unique value by increasing the current *timestamp* value. As noted before, we use the timestamp to index the transaction, as for example a transaction with timestamp i is denoted T_i. For two transactions T_i and T_j, if $j > i$ then T_i arrives before T_j. At any point of time, *liveT* is a special object

that contains a list of timestamps of read-only transactions that are currently pending.

Consider now read-only transactions. As shown in Algorithm 2, after a read-only transaction T_i arrives and gets a timestamp, it is added to the list of live transactions $liveT$. As we will show in the correctness analysis of our algorithm, read operations within a read-only transaction are K-legal; in other words, read-only transactions are K-legal. (In many systems, the kinds of transactions are identified at the beginning such as read balance and bank statements in bank systems, or product quantities in inventory systems). A read operation $x.r(data)$ in T_i tries to read the latest version by invoking function GetLatestVersion() (Algorithm 3). Algorithm 3 tries first to read $x.lastCommit$, the latest committed version. If the timestamp $x.lastCommit.ts$ is smaller than the transaction's T_i timestamp i, then GetLatestVersion() returns $x.lastCommit$. Otherwise, if $x.lastCommist.ts > i$, it finds a saved version v_j in $x.vl$ (version list) where $v_j.ts$ is the largest timestamp smaller than i. After that, it adds T_i timestamp to the version's rl (reader list) and returns the version to be read. When T_i finishes the execution of all its read operations, it calls TryC() (Algorithm 4) and commits directly (recall that in our algorithm read-only transactions do not abort).

FIGURE 5.2(a) shows an example where read-only transaction can read the version $x.lastCommit$ without violating the correctness of execution. The execution in FIGURE 5.2(a) illustrates the situation where there is no concurrent write on the object. The read-only transaction T_3 finds that the $x.lastCommit.ts$ equals to 2 which is smaller than its own timestamp 3. Therefore, it reads $x.lastCommit$ which shows the last write on the object x. In contrast, FIGURE 5.2(b) demonstrates a situation where there is one concurrent write on the object x. When the read-only transaction T_3 executes the operation $x.r(data)$ it finds that $x.lastCommit.ts = 4$

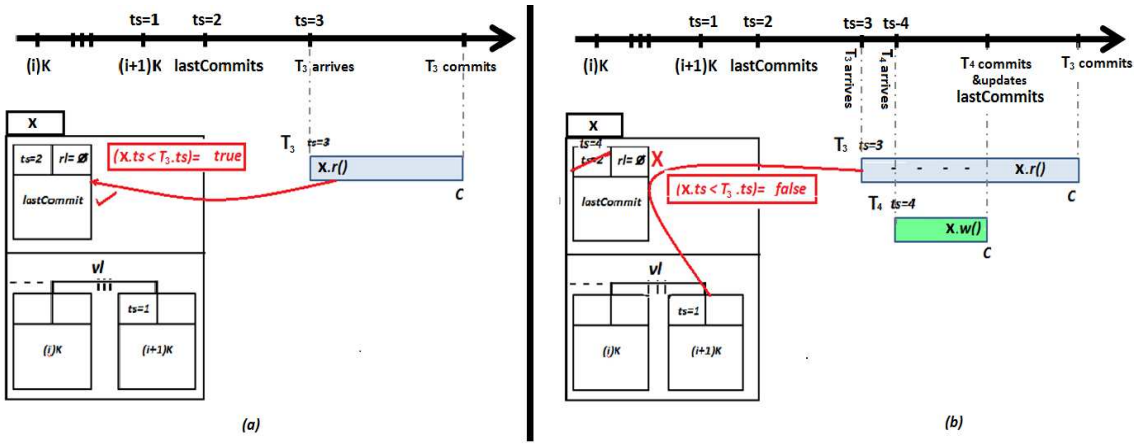

FIGURE 5.2. (a) Read-only Transaction with no Concurrent Writes, (b) Read-only Transaction with a Concurrent Write

(due to the update transaction T_4) which is greater than its own timestamp 3. Thus, it reads a version from $x.vl$. In particular, it reads the latest saved version with the timestamp smaller than its own.

We shift now our focus to update transactions. In Algorithm 2 consider an update transaction T_i and read operation $x.r(data)$. As before, the algorithm invokes GetLatestVersion() (Algorithm 3) but now it checks only $x.lastCommit$ and adds i to $x.lastCommit.rl$ (reader list). Then, Algorithm 1 checks the condition $x.lastCommit.ts > i$ and, if true, T_i aborts immediately, since the object has been updated by a concurrent transaction. Otherwise, it saves locally $x.lastCommit.ts$ (for validation) and reads the data of x. On the other hand, for any write operation, the transaction T_i just saves the newly written value to its local memory temporarily until it commits in shared memory; also T_i maintains its own $wSet$ (set of locally written objects) during the execution.

When the update transaction T_i finishes the execution of all operations, it calls TryC() (Algorithm 4). If any object x updated by T_i has been overwritten by a newer transaction than i, then T_i aborts. This is checked by invoking function Validate() (Algorithm 5), and checking if the timestamp $x.lastCommit.ts$ has a

value greater than i. Moreover, for all objects in T_i's $wSet$ (write set), we check if there is a transaction T_m that arrives after T_i (i.e. $m > i$) and reads the latest version, then it aborts. In this case, we may need to abort T_i because we will not be able to serialize T_i and T_m in the process of generating S which is required to prove the correctness of concurrent execution as we prove later. If T_i aborts, TryC() returns $false$ and then we change the *status* of T_i to *aborted*.

In commit of transaction T_i, TryC() locks[2] all objects in its $wSet$. We overwrite the $x.lastCommit$ version and update $x.lastCommit.ts = i$. We do not create new versions with each commit, but with each commit we increase the $x.versionCounter$ and we create a new version for x only every K commits. We let the new version's ts to be equal to the committed transaction's timestamp (i.e. i) and we add it to $x.vl$. After that, we release all locks and change the transaction *status* to *committed*. The garbage collection for unused object versions will be explained in detail in Chapter 8.

5.3 Correctness of the Algorithm

In the correctness analysis we prove that our algorithm is opaque for update transactions, and K-opaque for read-only transactions. Let H be an arbitrary execution history, and H' the respective complete history. Consider the sequential execution S which is a serialization of the transactions in H' such that the order of transactions is determined by the timestamps of the transactions, such that if in H' for any two transaction T_i and T_j, $i < j$, then $T_i <_s T_j$.

Lemma 5.1. *S preserves the real time order of H'.*

Proof. According to Algorithm 1, for a transaction T_i the timestamp i is obtained through an atomic operation $i \leftarrow timestamp$.getAndInc(); If $T_i <_{H'} T_j$ then, it

[2]To prevent deadlock we assume that all transactions acquire locks for the individual objects in a predefined order.

Algorithm 2: K-opaque Permissive Algorithm

```
/* K-opacity for read-only transaction */
/* Global variable initialization */
timestamp ← 0;
liveT ← ∅;

Upon receipt of a transaction do;

foreach transaction T_i do
    /* i gets a unique timestamp */
    i ← timestamp.getAndInc();
    T_i.status ← active;
    if T_i.kind = readonly then
        /* liveT is used for garbage collector */
        liveT ← i ∪ liveT;
    T_i.wSet ← ∅;
    while there is an unexecuted operation o do
        /* Read operation where y contains data */
        if o = x.r(y) then
            v ← GetLatestVersion(i, x);
            if v.ts > i then
                /* This check applies only for read operations within update
                transactions and can cause immediate abort */
                T_i.status ← aborted;
                return;
            // y contains the value that is read */ y ← v.data;
        else
            /* o = x.w(y); write local value y */
            x_local.data ← y;
            T_i.wSet ← x ∪ T_i.wSet;

    //Based on TryC(), transaction commits or aborts
    if TryC(i, i.wSet) then
        T_i.status ← committed;
    else
        T_i.status ← aborted;
    return;
```

Algorithm 3: GetLatestVersion(i, x)

```
last ← null;
Lock x;
if T_i.kind = readonly then
    if x.lastCommit.ts < i then
        last ← x.lastCommit;
        Add i to list x.lastCommit.rl;
    else
        v ← the most recent version in x.vl with timestamp smaller than i;
        last ← v;
        Add i to list v.rl;
else
    /* Update transaction */
    last ← x.lastCommit;
    Add i to list x.lastCommit.rl;
Unlock x;
return last;
```

has to be that $i < j$. Since S orders transactions in the timestamp order, then we also have that $T_i <_S T_j$, as needed. □

We continue to prove that S is K-legal with respect to any object x for read-only transactions, and then we prove that it is 1-legal for update transactions.

Lemma 5.2. *For any object x, the history S is K-legal with respect to read-only transactions accessing x.*

Proof. Let T_i be a read-only transaction. Note that in our algorithm read-only transactions do not abort, and hence T_i does not abort. Suppose T_i executes operation $x.r_i(y)$. According to function GetLatestVersion(), we have that T_i observes either $x.lastCommit.ts < i$ or $x.lastCommit.ts > i$. We examine these two cases separately.

i. $x.lastCommit.ts < i$:

then GetLatestVersion() returns $x.lastCommit$ and data $y = x.lastCommit.data$,

Algorithm 4: TryC$(i, i.wSet)$

```
/* Check if T_i is readonly */
if T_i.kind = readonly then
    /* Remove T_i from liveT */
    liveT ← liveT \ i;
    return true;
/* T_i has to be update transaction */
L ← ∅;
/* Assume a predetermined order for the objects */ forall the x ∈ T_i.wSet
do
    Lock x;
    L ← L ∪ x;
    if Validate(i, x) = false then
        unlock all locked objects in L;
        return false;
forall the x ∈ i.wSet do
    x.versionCounter.getAndInc();
    if x.versionCounter mod K = 0 then
        /* Add new version to x.vl */
        Add (i, x_local.data, nil) to x.vl;
    /* Overwrite x.lastCommit */
    x.lastCommit ← (i, x_local.data, nil);
Unlock all objects in L;
return true;
```

Algorithm 5: Validate(i, x)

```
/* Check if lastCommit has been overwritten */
if x.lastCommit.ts > i then
    return false;
/* Check if some other transaction T_m has read the latest version where
m > i */
if x.lastCommit.rl contains a transaction T_m, where m > i then
    return false;
return true;
```

which is the latest version of the object at that moment when x is accessed by T_i. Let T_j be the transaction that committed the value; that is, $j = x.lastCommit.ts < i$. Since S preserves the timestamp order, T_j appears before T_i in S. Suppose that there is another transaction T_k, with $j < k < i$, that commits a value to object x. If T_k commits after $x.r_i(y)$ locks x (in GetLatestVersion()), then according to function Validate(), T_k has to abort because T_i is in the reader list of x when T_k attempts to commit. On the other hand, if T_k commits before $x.r_i(y)$ locks x, then T_i must have read the value committed by T_k, or in other words $T_k = T_j$.

ii. $x.lastCommit.ts > i$:

then GetLatestVersion() returns a version v, and data $y = v.data$, where v belongs to version list $x.vl$ and it is the latest version of x with timestamp $v.ts = j < i$. Let T_j be the transaction that created version v. We need to prove that there cannot be more than $K - 1$ other committed transactions for object x between the time that T_j commits and T_i starts in H'. We observe that any transaction T_k that commits a value for x after T_j must have timestamp $k > j$, since otherwise the interval of T_k would contain the interval of T_j in H', and according to function Validate() T_k would abort. We have that in S, T_j appears before T_i, since $j < i$. Let X be the set of transactions which appear in S between T_j and T_i and commit a value for x (for any $T_k \in X$ it holds that $j < k < i$). We want to show that $|X| \leq K - 1$. Similar to the reasons explained above in case i, X cannot contain any transaction which commits after $x.r_i(y)$ locks x. Moreover, X cannot contain any transaction T_k that commits before T_j, since in H' interval T_j would contain interval T_k and according to function Validate() T_j would

abort. Hence, all the transactions in X must have timestamp greater than j and must commit in H' after T_j. If $|X| \geq K$, according to our algorithm, a newer version v' of x must have been saved (in $x.vl$) after T_j commits and before T_i starts, by some transaction $T_\zeta \in X$. However, this is impossible, since T_i would have read v' and not v.

Therefore, we have that in case i execution S is 1-legal, while in case ii the execution S is K-legal. □

Lemma 5.3. *For any object x, the history S is 1-legal with respect to update transactions accessing x.*

Proof. Now consider the update transactions that access object x. Let T_i be an update transaction that invokes operation $x.r_i(y)$. According to the algorithm, if $x.lastCommit.ts > i$, then T_i is aborted and operation $x.r_i(y)$ never completed. On the other hand, if $x.lastCommit.ts < i$ then $x.r_i(y)$ completes with $y = x.lastCommit.data$. Let $j = x.lastCommit.ts$, namely, T_i reads the value written by T_j, with $j < i$. Similar to the proof of Lemma 5.2, any transaction T_k that commits a value for x after T_j must have timestamp $k > j$.

In S transaction T_j appears before T_i. We only need to show that in S there is no other committed transaction between T_j and T_i for object x. Suppose that there is a transaction T_k, with $j < k < i$, which appears between T_j and T_i in S and commits a value to x. If T_k commits before T_j in H', function Validate() would cause to abort T_j. If T_k commits before $x.r_i(y)$ locks object x, then T_i must have used the value committed by T_k. On the other hand, if T_k commits after $x.r_i(y)$ locks object x, then according to function Validate() T_k has to abort, since T_i is in the reader list of x when T_k attempts to commit, and $i > k$. □

Since H' respects the real time order of H, considering all objects used in H, from Lemmas 1, 2 and 3 we obtain the following theorem.

Theorem 5.4. *Any execution history H of our algorithm is K-opaque with respect to read-only transactions and* 1*-opaque with respect to update transactions.*

Theorem 5.5. *An approximately opaque multi-version transactional memory is never to abort read-only transaction but update transaction is aborted only if it has a conflict with another transaction.*

Proof. Clearly read-only transactions execute the read operations and in TryC() they commit directly. However, update transaction in TryC() may abort if it invalidates the correctness of another transaction. So it has to lock the object and check $x.rl$ to know if it is being read by another transaction that has greater timestamp. In such a case the update transaction aborts. This obviously means it aborts to allow another transaction to commit. □

It is easy to check that the proposed algorithm does not deadlock, since function GetLatestVersion() accesses one object at a time, and function TryC() accesses objects in a predetermined order, avoiding racing situations.

Lemma 5.6. *Our algorithm does not deadlock.*

Assume that the transactions in our algorithm access the set of objects $O = (x_1, x_2, ..., x_n)$. Let V be the set of all committed versions (updating $x_i.lastCommit$) and V' the set of all saved versions (saved in $x_i.vl$).

Theorem 5.7. *In any execution of our Algorithm, the total number of saved object versions is $|V'| = \Theta(|V|/K + |O|)$.*

Proof. Through the execution, for any object x_i let the number of committed versions be v_{x_i} (updating $x_i.lastCommit$). The total number of committed versions

for all objects is $|V| = \sum_{i=1}^{n} v_{x_i}$. Our algorithm saves a new version for object x_i each K object commits. Thus, the total number of saved versions for object x_i (saved in $x_i.vl$) is v_{x_i}/K, and consequently, the total number of saved versions $|V'|$ for all objects will be $|V|/K$. In addition, each object has a *lastCommit* version which adds a number of $|O|$ versions to $|V|/K$. □

Now, if we exclude the last committed versions, the total version space of regular multi-version permissive transactional memory is $\Theta(|V|)$, since every version is saved at some point of time. On the other hand, from Theorem 5.7, with our approximately opaque multi-version permissive algorithm we only create a new version each K commits reducing this number to $\Theta(|V|/K)$. Furthermore, with the garbage collector, old versions are deleted which reduces the active number of $|V'|$ at any point of the execution, and that will be shown in detail in chapter 8.

5.4 Experimental Results

In our experiment, we simulate Bank, Linked-list and Red-black Tree benchmarks from TinySTM-1.0.5 [13], but we modify the structure of the object to match our specification. We run the experiments on a machine with dual Intel(R) Xeon(R) CPU E5-2630 (6 cores total) clocked at 2.30 GHz. Each run of the benchmark takes about 5500 milliseconds using 10 threads. In the Bank benchmark, there are three kinds of operations which are read balance, write amount and transfer. In the Linked-list and the Red-black Tree benchmarks, we have search operations, add node and delete node. Read balance (in Bank) and search (in Linled-list and Red-black Tree) are read-only, but write, transfer (in Bank) and add/delete node (in Linled-list and Red-black Tree) are update transactions. In our execution we generate 50% read-only transactions and 50% update ones.

In FIGURE 5.3, we compare the throughput (commits per time) of an opaque execution (1-opaque), 2-opaque, 4-opaque and 8-opaque using the three bench-

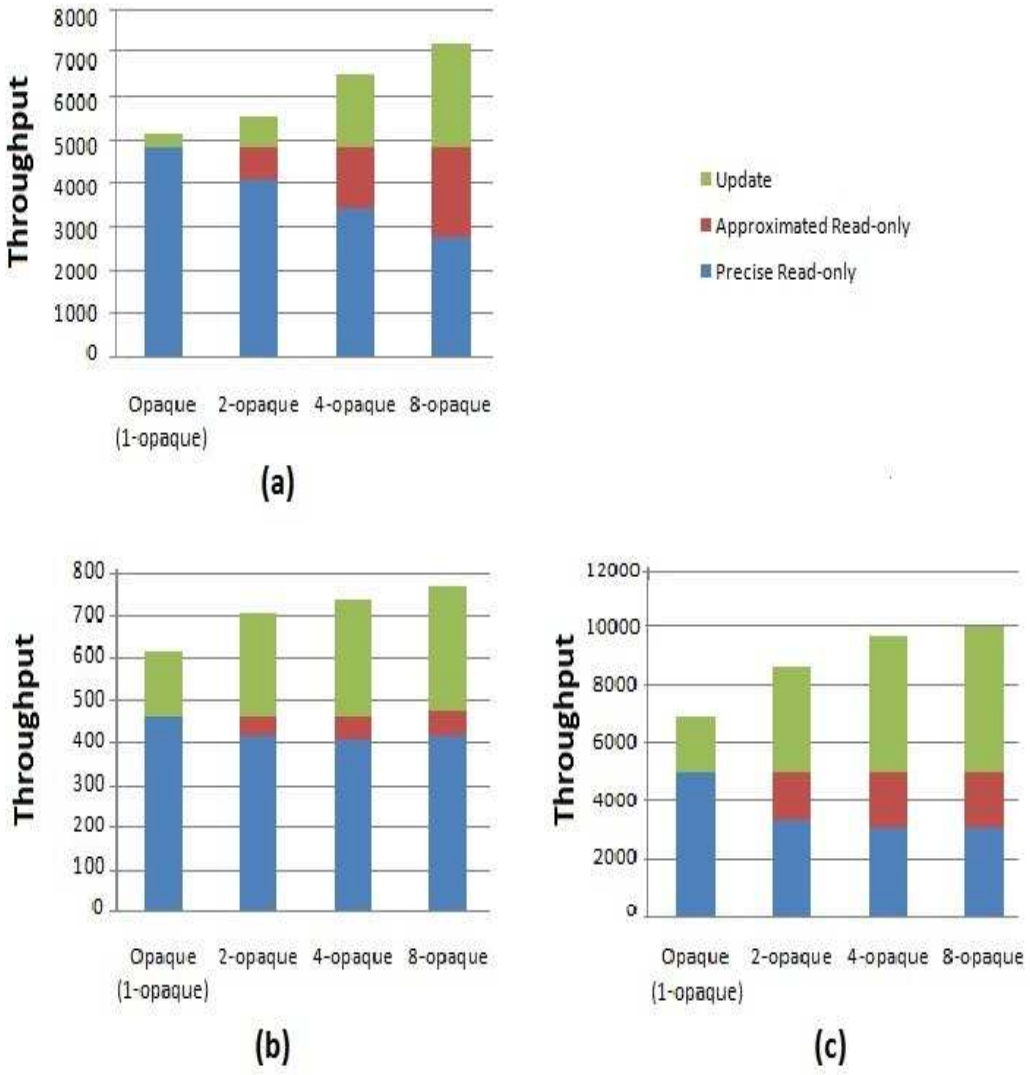

FIGURE 5.3. (a) Comparison of the Throughputs (Committed Transactions Per Time) of Opaque, 2-opaque, 4-opaque and 8-opaque Using the Linked-list Benchmark. (b) Comparison of the Throughputs of Opaque, 2-opaque, 4-opaque and 8-opaque Using the Red-black Tree Benchmark, (c) Comparison of the Throughputs (Committed Transactions Per Time) of Opaque, 2-opaque, 4-opaque and 8-opaque Using the Bank Benchmark.

marks. Clearly, the relaxed opacity in 2-opaque, 4-opaque and 8-opaque help to avoid some aborts and to improve the throughput. Furthermore, in 1-opaque all read-only transactions are precise but in 2-opaque, 4-opaque and 8-opaque the percentage of approximated read-only transactions is smaller than the percentage of the precise ones. We note that there is an increase in the number of commit-

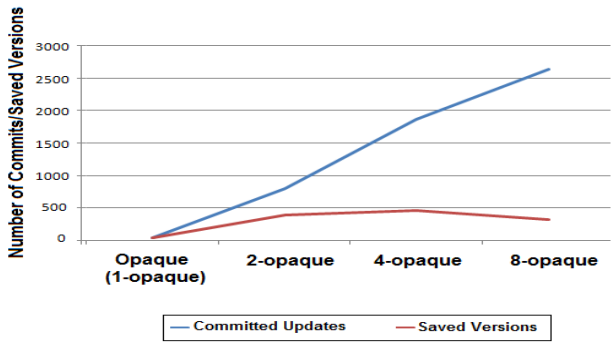

FIGURE 5.4. Compare the Number of Committed Updates to the Number of the Saved Versions in Opaque, 2-opaque, 4-opaque and 8-opaque Using the Linked-list Benchmark

ted updates since relaxing the opacity of a read-only transaction sometimes allows to avoid many aborts; as 1-opaque read-only transaction may conflict with many update ones.

FIGURE 5.4 shows a comparison between the number of committed updates and the number of the saved versions using the Linked-list benchmark. In 1-opaque the number of committed updates and the number of the saved versions are the same, since we save a new version with each committed update. In 2-opaque and 4-opaque, the number of saved version increases because such relaxations allow to commit very large number of updates. However, in 8-opaque the number of committed updates increases but the number of none-saved versions is very large (as we save 1 version every 8 commits), so the number of saved versions decreases.

Chapter 6

Approximately Opaque Transactional Memory for Read/Write and Count Objects

6.1 Introduction

After the promising results of applying approximated opacity on read-only transactions, in this chapter we apply the concept of K-opacity on update transactions as well. The update transactions access regular read/write and count objects, which are common objects used in typical concurrent programs. This chapter shows the design of the K-opaque algorithm for read/write and counting operations. In fact, for counting operations we just consider addition and subtraction. Then we show the proof for correctness. After that, we present some experimental results to illustrate the impacts of consistency relaxation on performance.

6.2 The Design of the Algorithm

Let us start with the structure of the TM objects where there are two kinds of objects which are read/write and count objects. FIGURE 6.1 (a) shows the read/write object x has fields of $(ts, data, commitsCounter, maxT, dataT, rl[])$, where ts shows the maximum timestamp of the transactions that commit values to x, $data$ is the value of x, $commitsCounter$ records the number of commits on x to ensure that we overwrite x every K commits, $maxT$ records the maximum timestamp of the transactions that commit on x, $dataT$ records the data of $maxT$ transaction and $rl[]$ records the timestamps of the live transactions that have been reading x.

FIGURE 6.1 (b) illustrates the count object x which consists of some fields which are $(ts, data, wl[][], index)$, where ts represents the maximum timestamp of the transactions that commit values to x, $data$ is the value of x and $wl[][]$ is a writer list which is an array of size n, where n is the number of threads in

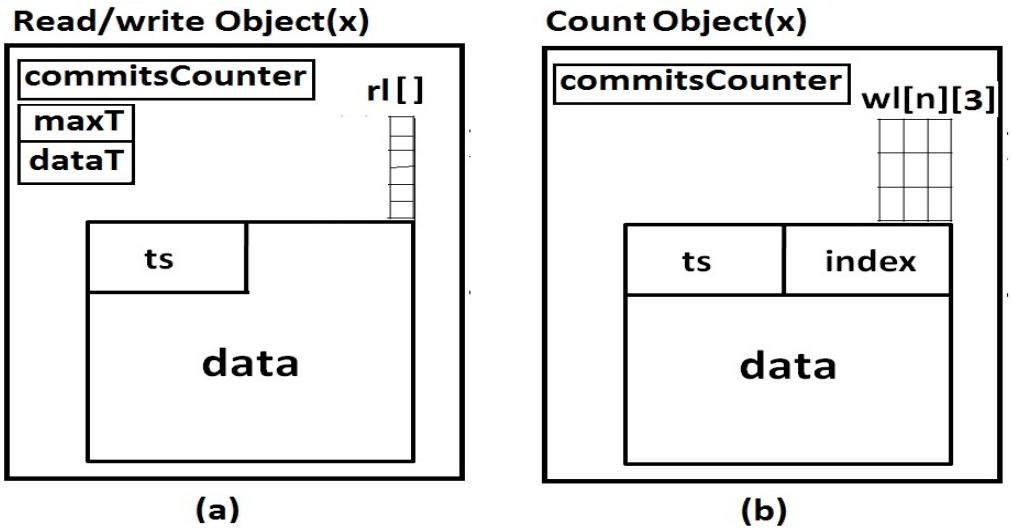

FIGURE 6.1. (a) Read/Write Object Data Structure. (b) Count Object Data Structure.

the system. Actually, $wl[][]$ has three columns such that each update transaction writes to x must post its timestamp (tx) in the first column, the *data* it writes in the second column, and the transaction status ($live, committed$ or $aborted$) in the third column. Since the $wl[][]$ is an integer array we represent the transaction status with 0, 1 and 2 corresponding respectively to $live, committed$ and $aborted$. In addition, we use $index$ to tell the update transactions where to write in $wl[][]$.

To start with our algorithm, the classes RWObject and Count, show the structure of the objects. Basically, our algorithm is timestamp-based that executes transactions concurrently but it does not update objects with each commits. Indeed every K commits on the object overwrites it with a new value.

Main() (Algorithm 6) shows the structure of the transaction where it has a unique timestamp i using getAndIncerement() method that increments the value of a special object *timestamp* by 1. Transaction has a status $T_i.status$ that is set to $live$ and adds itself to the $liveT$ list which is used for the garbage collector in chapter 8. It also has an access set $T_i.accSet$ that is maintained during the execution. Moreover, *valid* Boolean variable is used as a flag to continue the execution of T_i

or abort it immediately when it is needed. We have also two global variables which are *correct* (that is used to calculate the correct value of update transactions) and *commits* which counts the number of commits on x.

Then, transaction T_i executes all its operations which are read, write and add. For all operations we add the object x to $T_i.accSet$ and then execute the operation by calling function ReadWrite() or Add(). If the functions return $false$, then we change *valid* to $false$ and T_i aborts immediately. When T_i executes all operations it calls TryC() and based on that it commits or aborts.

6.2.1 Read/Write Operations

For read/write operations, in ReadWrite() (Algorithm 7), if the transaction's timestamp i is smaller than $x.ts$, then T_i aborts since it violates the correctness property (timestamp-based execution). Otherwise, if $data = Null$ means the operation is read, then we add i to $x.rl[]$ and get $x.data$. At the end of T_i execution, it calls TryC() (Algorithm 10) where the read operations do not validate.

On the other hand, if $data\ != Null$ means the operation is write, T_i writes in its own local memory. At the end of T_i execution, it calls TryC() to validate. It locks x (we lock objects in a predetermined order to avoid deadlock) to check if there is another transaction with a timestamp greater than i commits on x. Also, it checks if another transaction with greater timestamp have read x. In these two cases T_i aborts. Otherwise if the validation goes well, in Commits() (Algorithm 12) we change $T_i.status$ to *committed* and we store the maximum timestamp of the transactions that commit on x in $x.maxT$, and the data of that transaction in $x.dataT$. Then we increase $x.commitsCounter$ by one. If it is equal to K, then we overwrite x with the timestamp and the data that are stored in $x.maxT$ and $x.dataT$. We reset $x.commitsCounter$, $x.maxT$ and $x.dataT$, and we release all

locks. However if $x.commitsCounter < K$, T_i just commits and does not overwrite x.

In Abort() (Algorithm 11), we change $T_i.status$ to *aborted*.

6.2.2 Counting Operations

For counting operations, in Add() (Algorithm 8), if $x.ts > i$, then T_i aborts. Otherwise, to read the count object x ($x.add(0)$) we get $x.data$ and write locally. To add a nonzero value to x, if T_i accesses x for the first time, then T_i adds itself to $x.wl[][]$, such that it increases $x.index$ which tells in which row T_i can write, and then it posts i, *data* and $T_i.status$ to $x.wl[][]$. T_i also writes in its local memory. If T_i already accessed x, then it would already be in $x.wl[][]$. In this case, just add the data of the operation to the data that is already stored in $x.wl[][]$.

Then, in TryC(), we lock x and call CheckStatus() (Algorithm 9) to know if T_i is aborted by another transaction, or x has been overwritten by a transaction T_m which has a timestamp greater than i, then we return $false$ and T_i aborts.

Otherwise the validation goes well and then T_i calls Commit() and checks if the number of committed transactions including (T_i) equals to K, then T_i overwrites x with the correct value that considers all values of the committed transactions in $x.wl[][]$. Therefore, T_i aborts all *live* transactions in $x.wl[][]$, copies the data of *committed* transactions from $x.wl[][]$ to temporary array $temp[][]$ and sorts $temp[][]$ based on the transactions' timestamps. After that, it calculates the *correct* value of the $x.data$ and finds the maximum timestamp in $temp[][]$ (max). It overwrites x with $x.ts = max$, $x.data = correct$ and resets all other fields of x. We release all locks and change $T_i.status$ to *committed*. However, if the number of committed transactions is less than K, T_i just commits and changes its status in $wl[][]$.

In Abort() (Algorithm 11), we change $T_i.status$ to *aborted*, and we change T_i status to aborted in the writer lists of all objects in $T_i.accSet$.

```
Class RWObject
int ts ← 0;
int data ← 0;
int commitsCounter ← 0;
int maxT ← −1;
int dataT;
int rl[];
```

```
Class Count

int ts ← 0;
int data ← 0;
int commitsCounter ← 0;
int index ← 0;//Array's index
//wl[][] is an array to record update transactions
//n is the number of the thread in the system
int wl[n][3];
```

6.3 Correctness of the Algorithm

In the correctness analysis we prove that our algorithm is K-opaque for all transactions. Let H be an arbitrary history of an execution. Let H' be a complete history that we obtain such that if a pending transaction in H didn't invoke either Commit() or Abort() then its status is aborted, while in any other case the status is either committed or aborted, according to which of the two functions the transaction invoked. Let S be the sequential execution, which is a timestamp-based serialization of the transactions in H'.

Lemma 6.1. *S preserves the real time order of the transactions in H'.*

Proof. According to Main() (Algorithm 6), each transaction T_i obtains a unique timestamp using $i \leftarrow$ *timestamp*.getAndInc(), which is an atomic operation. If $T_i <_{H'} T_j$, then $i < j$. Since S orders transactions based on their timestamp, we get $T_i <_S T_j$. In other words, $<_{H'} \subseteq <_S$, as needed. □

Algorithm 6: Main ()

```
/* Global variable initialization */
timestamp ← 0;
int correct ← 0;//To calculate the correct value
int commits ← 0;
bool valid = true;

Upon receipt of a transaction do;

foreach (transaction T) do
    //Get a unique timestamp
    i ← Timestamp.getAndInc();
    T_i.status ← live;
    /* liveT is used for garbage collector */
    liveT ← i ∪ liveT;
    T_i.accSet ← ∅;
    while (there is unexecuted operation on object x) do
        switch operation on x do
            case (x.r())
                //Read operation
                T_i.accSet ← x ∪ T_i.accSet;
                valid = ReadWrite(i, x, data);
            case (x.w(data))
                //Write operation
                T_i.accSet ← x ∪ T_i.accSet;
                valid = ReadWrite(i, x, data);
            case (x.add(data))
                //Add operation
                T_i.accSet ← x ∪ T_i.accSet;
                valid = Add(i, x, data);
        if (valid = false) then
            Abort(i, T_i.accSet);
            return;

    //Based on TryC(), transaction commits or aborts
    if (TryC(i, T_i.accSet)) then
        Commit(i, T_i.accSet);
    else
        Abort(i, T_i.accSet);
    return;
```

Algorithm 7: ReadWrite($i, x, data$)

```
//If a newer transaction overwrites x, then T_i aborts
if (i < x.ts) then
    return false;
if (data = Null) then
    //Read operation
    add i to x.rl[];
    get(x.data);
else
    //Write operation
    //Write the value data to x in local memory
    let x_local.data ← data;
return true;
```

We continue to prove that S is K-legal with respect to any object x for any transaction.

Lemma 6.2. *The history S is K-legal, for any read/write object x.*

Proof. Let T_i be a transaction that executes read operation $x.r_i(data)$. According to function ReadWrite(), T_i checks $x.ts$ that shows the timestamp of the last transaction that overwrites x. If $x.ts = j > i$, then T_i aborts. If T_j is the last transaction that overwrites x, then we need to prove that there cannot be more than $K-1$ other committed transactions on x between the time that T_j commits and T_i performs its read in S. Since $j < i$ we have that in S, $T_j <_S T_i$. Let Q be the set of transactions which appear in S between T_j and T_i which have a write operation to x and commit (Q does not contain T_j or T_i). We only need to show that $|Q| \leq K-1$.

We first show that none of the transactions in Q overwrite x. Suppose for the sake of contradiction that there is a transaction $T_m \in Q$ which overwrites x, namely, it sets $x.commitsCounter = 0$ and updates $x.data = x_{local}.data$ and $x.ts = m$. Note that $j < m < i$. We examine three cases with respect to when T_m commits in H':

Algorithm 8: Add$(i, x, data)$

```
int max ← −1;
//If a newer transaction overwrites x, then T_i aborts
if (x.ts > i) then
    return false;
//If the operation is reading the counter
if (data = 0) then
    //Get the value of x
    int y = x.data;
    let x_local.data ← y;
else
    //If the operation is writing to the counter
    //If the same transactions execute more than one counting operation on
    the same object x
    for ( from m ← 0 to m = n) do
        if (wl[m].tx = i) then
            x.wl[].data = x.wl[].data + data;
    //If T_i accesses x for first time
    r ← x.index.getAndInc();
    //wl is an array of size n
    x.wl[r].tx ← i;
    x.wl[r].data ← data;
    x.wl[r].status ← T_i.status;
    //Write the value data to x in local memory
    int y = x.data;
    let x_local.data ← y;
return true;
```

- T_m commits before T_j commits.

  In this case, when T_j invokes TryC() it observes one of the following two scenarios:

  - $maxT = m > j$: T_j observes that a transaction T_m with higher timestamp ($m > j$) has committed on x (but it does not overwrite x), since $maxT$ records the maximum timestamp of the committed transactions, and hence T_j aborts.

Algorithm 9: CheckStatus(i, x)

```
int T_iRemoved ← 0;
//Check if T_i is aborted and removed from wl by another transaction
for ( from m ← 0 to m = n) do
    if (x.wl[m].tx = i) then
        T_iRemoved ← 1;//It is not removed
        break;
//If T_i is removed from wl
if (T_iRemoved = 0) then
    commits = −1;
    return commits;
//T_i still in wl
x.wl[m].status = committed;
//Check how many committed transactions
commits = x.commitsCounter;
return commits;
```

- $x.ts \geq j$: T_j observes that x was actually overwritten by T_m or by a more recent transaction, and hence, T_j aborts.

In either scenario, T_j aborts, which is impossible.

- T_m commits and overwrites x before $x.r_i()$ starts.

  In this case, T_i reads either the value written by T_m or by a more recent transaction (with timestamp $x.ts \geq m$). However, this contradicts the assumption that T_i reads $x.ts = j$.

- T_m commits and overwrites x after $x.r_i()$ ends.

  In this case, in its TryC() transaction T_m will observe that $x.commitsCounter \geq K - 1$ (which means T_m is the K^{th} transaction), and also it observes that T_i is in the reader list of x (that is, $i \in x.rl$ with $i > m$), and the combination of these two observations together force T_m to abort, which is a contradiction.

Therefore, no transaction in Q overwrites x. This implies that each transaction in Q increments $x.commitsCounter$. For a transaction $T_k \in Q$, let c_k denote the

Algorithm 10: TryC($i, T_i.accSet$)

```
L ← ∅;
forall the (x in T_i.accSet) do
    lock();
    L ← L ∪ x;
    switch Type of x do
        case x is RWObject
            if (data = Null) then
                //Read operations do not validate
            else
                //For write operation
                if ((x.ts > i) ∨ (maxT > i)) then
                    //Aborting because a concurrent write
                    Unlock() all objects in L;
                    return false;
                if ((x.rl[] has a transaction T_m where
                m > i) ∧ (x.commitsCounter = K − 1)) then
                    //Aborting because a concurrent read
                    Unlock() all objects in L;
                    return false;
        case (x is Counter)
            commits ←CheckStatus(i, x);
            if ((commits = −1) ∨ (x.wl[] has a transaction T_m where m > i))
            then
                Unlock() all objects in L;
                return false;
return true;
```

Algorithm 11: Abort($i, T_i.accSet$)

```
T_i.status ← aborted;
//Change its status in all counters and queues it accesses
forall the (x in T_i.accSet) do
    x.wl[x.index].status ← aborted;
    remove i from x.rl[];
return;
```

Algorithm 12: Commit($i, T_i.accSet$)

```
int max ← −1;
T_i.status ← committed;

forall the (x in T_i.accSet) do
    switch Type of x do
        case x is RWObject
            if (x.maxT < i) then
                //Recording the maximum timestamp and its data
                x.maxT = i;
                x.dataT = data;
            x.commitsCounter++;
            if (x.commitsCounter = K) then
                //Overwrite x
                x.ts = x.maxT;
                x.data = x.dataT;
                reset x.commitsCounter, x.maxT and x.dataT;
        case (x is Counter)
            x.commitsCounter++;
            if (x.commitsCounter = K) then
                //Calculate the correct value considering only the committed
                transactions
                abort the other live transactions in x.wl[][];
                copy the data of committed transactions from x.wl[][] to
                temp[][];
                sort(temp[][]); //Based on the timestamps
                max = max(temp[].tx); //Maximum timestamp in temp[][]
                for (from j ← 0 to j < K) do
                    correct ← correct + temp[j].data;
                x.data ← correct;
                x.ts ← max;
                reset x.wl[][] and x.index;
            else
                //Just Commit
                x.wl[].status = Committed;

Unlock() all objects in L;
return true;
```

respective updated value of $x.commitsCounter$. Next we show that for any pair $T_k, T_l \in Q$, where $k < l$, it must hold that $c_k \neq c_l$. Since $x.commitsCounter$ is updated atomically (is locked by each transaction that modifies it), if $c_k = c_l$ then some transaction T_m must commit and overwrite x (reset $x.commitsCounter$) after T_k commits and before T_l commits in H'. We know that transaction T_m cannot be in Q. Therefore, $m < j$, which is impossible since T_m would abort observing a higher timestamp on x than its own ($x.ts \geq j$). Therefore, the transactions in Q assign unique values to $x.commitsCounter$. Since the $x.commitsCounter$ cannot exceed $K - 1$ and none of the transactions in Q set $x.commitsCounter = 0$, we get that $|Q| \leq K - 1$, as needed.

As a special case, the same properties for Q hold even if T_i reads the initial value of x and T_j is replaced by a special instantaneous event that initializes x. □

Lemma 6.3. *The history S is K-legal, for any count object x.*

Proof. According to the implementation of Algorithm Add(), within a transaction T_i we can treat the sequence of all add operations as a single $x.add_i(v_i)$ operation which aggregates the added values of the operations to a single operand value v_i, to be added to the current value of x. So assume that in each transaction T_i there is at most one add operation on object x with operand v_i.

Let $S(x) = T_1, T_2, \ldots, T_q$ be the subsequence of transactions in S that invoke an add operation to x. A transaction T_i that overwrites x is a transaction that commits and updates $x.data = x_{local}.data$, and it also sets $x.commitsCounter = 0$ and $x.ts = i$. Let $S'(x)$ denote the subsequence of $S(x)$ consisting of all transactions that commit. Let $S''(x)$ denote the subsequence of $S(x)$ consisting of all transactions that overwrite x.

Let $r_1, \ldots, r_q$ be the respective sequence of returned values of the add operations of transactions in $S(x)$, namely, $r_i = x.add_i(v_i)$ is the value returned by T_i. Let r'_g denote the *precise value* that would have been returned by the add operation of transaction T_g when the consistency is precise, that is, $r'_g = v_0 + \sum_{(1 \leq l < g) \wedge T_l \in S'(x)} v_l$, where v_0 is the initial value of x. Let $o_g = r_g + v_g$ be the result of the add operation of transaction T_g. Similarly, let $o'_g = r'_g + v_g$ be the respective precise result of the add operation of transaction T_g.

For each $T_g \in S(x)$ let P_g denote the set of the last K transactions which precede T_g in $S'(x)$. We only need to prove the following two properties:

(i) For each $T_g \in S''(x)$, $r_g = r'_g$ (the transactions in $S''(x)$ are precise).

(ii) For each $T_g \in S(x) \setminus S''(x)$, either $r_g = o_l$ and $T_l \in P_g$ and $T_l \in S''(x)$, or $r_g = v_0$ and $|P_g| < K$.

We prove these properties by induction on q. For $q = 0$, $S(x)$ is empty and the properties hold trivially. Assume now that the properties hold for any $q < i$; we will show that the properties hold also for $q = i > 0$.

Consider now the last transaction T_i. According to function Add(), T_i checks $x.ts$ that shows the timestamp of the last transaction that overwrites x. Suppose that $x.ts = j > i$. Let Q be the set of transactions which appear in S between T_j and T_i and that have an add operation to x and commit (Q does not contain T_j or T_i). We continue to show that $|Q| \leq K - 1$.

We first show that none of the transactions in Q overwrite x. Suppose for the sake of contradiction that there is a transaction $T_m \in Q$ which overwrites x. Note that $j < m < i$. We examine three cases with respect to when T_m commits in H':

- T_m commits before T_j commits.

  In this case, when T_j invokes TryC() it observes one of the following two scenarios:

  - $m \in x.wl$ and $m > j$: T_j observes that a transaction T_m with higher timestamp ($m > j$) has committed on x but it does not overwrite x, and hence T_j aborts.
  - $x.ts \geq j$: Since T_m commits and overwrites x based on the Algorithm Commit(), T_m aborts all live transactions with smaller timestamp than m, and hence T_j aborts.

  In either scenario, T_j aborts, which is impossible.

- T_m commits and overwrites x before $x.add_i()$ starts.

  In this case, T_i reads either the value of x written by T_m or by a more recent transaction (with timestamp $x.ts \geq m$). However, this contradicts the assumption that T_i reads $x.ts = j$.

- T_m commits and overwrites x after $x.add_i()$ ends.

  In this case, in its TryC() transaction T_m will observe that $x.commitsCounter \geq K-1$ (which means T_m is the K^{th}th transaction), and also it observes that T_i is in the writer list of x (that is, $i \in x.wl$ with $i > m$), and the combination of these two observations together force T_m to abort, which is a contradiction.

Therefore, no transaction in Q overwrites x. This implies that each transaction in Q increments $x.commitsCounter$. Therefore, similar to the proof of Lemma 6.2 the transactions in Q assign unique values to $x.commitsCounter$. Since the $x.commitsCounter$ cannot exceed $K-1$ and none of the transactions in Q set $x.commitsCounter = 0$, we get that $|Q| \leq K-1$. The same observations for Q

hold even if T_i reads the initial value of x and T_j is replaced by a special event that initializes x.

Suppose now that $T_i \in S''$. When T_i invokes Algorithm Commit() the only committed transactions in the writer list of x are the ones in set Q. Therefore, the value returned by the add operation of T_i is equal to $r_i = o_j + \sum_{T_l \in Q} v_l$. By induction hypothesis, $r_j = r'_j$, and hence $r_i = r'_i$, and therefore, property (i) holds. If $T_i \in S \setminus S''$, then it returns $r_i = r_j = r'_j$. Since, $Q \cup \{T_j\} \subseteq P_i$ property (ii) holds as well. (Note that properties (i) and (ii) hold even if T_i reads the initial value v_0 and T_j is replaced with a special initialization event of x.) □

Based on Lemmas 6.1, 6.2 and 6.3, we obtain the following theorem.

Theorem 6.4. *Any execution history H of our algorithm is K-opaque.*

6.4 Experimental Results

In our experiment, we simulate Bank from TinySTM-1.0.5 [13], but we modify the structure of the object and the algorithm to match our specifications. We run the experiments on a machine with dual Intel(R) Xeon(R) CPU E5-2630 (6 cores total) clocked at 2.30 GHz. Each run of the benchmark takes about 7000 milliseconds using 10 threads. We test the benchmark to compare the opaque execution (1-opaque) with 2-opaque, 4-opaque and 8-opaque. The the Bank benchmark is used to test the counting operations. In the Bank benchmark, each account is a count object. Also there are three kinds of operations which are read balance ($add(0)$), write amount ($add(data)$ where $data$ is a positive or negative number), and transfer (which has two add operations one to subtract a number from one account and add it to another account). Read balance represents read-only transaction, but write and transfer are update transactions. We run our experiment with different ratios of read-only and update transactions. We run our execution

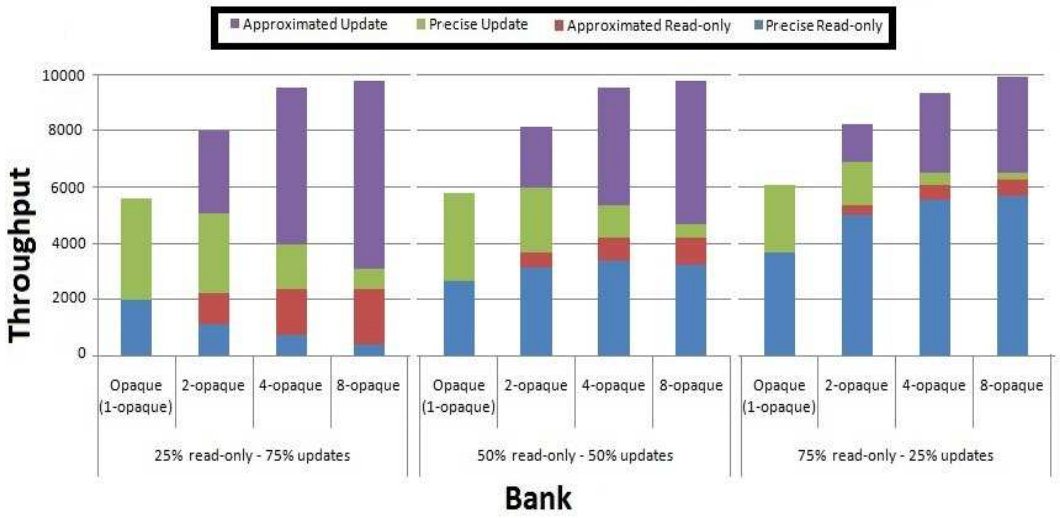

FIGURE 6.2. Compare the Throughput of 1-opaque, 2-opaque, 4-opaque and 8-opaque Executions on the Bank Benchmark with 25% Read-only and 75% Update, 50% Read--only and 50% Update, and 75% Read-only and 25% Update Transactions.

with 25% read-only and 75% update, 50% read-only and 50% update, and 75% read-only and 25% update transactions.

FIGURE 6.2 shows the throughput of 1-opaque, 2-opaque, 4-opaque and 8-opaque executions on the Bank benchmark. The figure shows an execution of 25% read-only and 75% updates, where in opaque executions all committed transactions are precise. In 2-opaque, 4-opaque and 8-opaque executions the throughput increases because of the relaxation (which means to have some approximated transactions) of some read-only and update transactions results in the avoidance of some aborts. Moreover, the same thing happens with executions of 50% read-only and 50% updates, and 75% read-only and 25% update transactions.

FIGURE 6.3 shows the aborts rate of 1-opaque, 2-opaque 4-opaque and 8-opaque executions on the Bank benchmark. The figure shows executions of 25% read-only and 75% updates, 50% read-only and 50% updates, and 75% read-only and 25% update transactions. In all executions the aborts rate decreases as we relax the

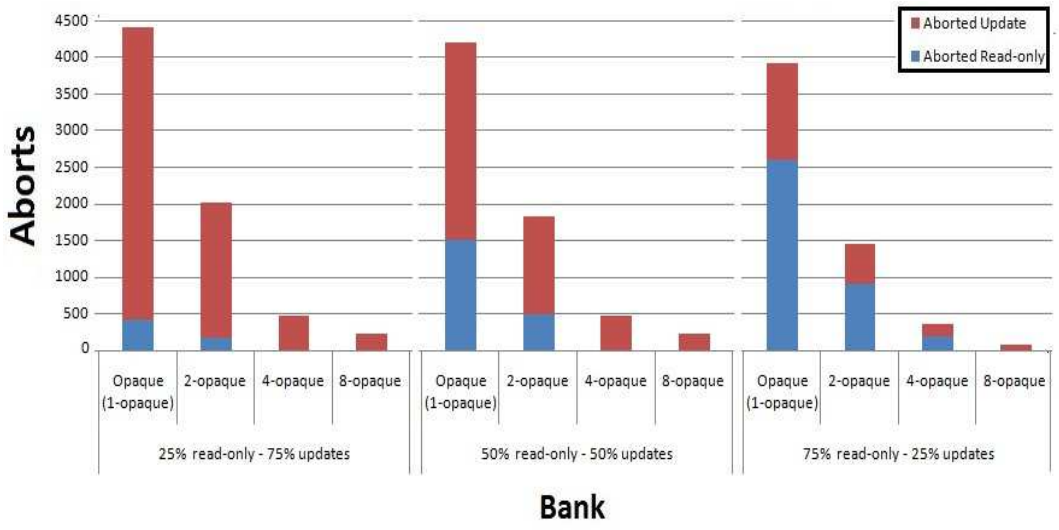

FIGURE 6.3. Compare the Aborts of 1-opaque, 2-opaque, 4-opaque and 8-opaque Executions on the Bank Benchmark with 25% Read-only and 75% Update, 50% Read-only and 50% Update, and 75% Read-only and 25% Update Transactions.

opacity. In 8-opaque, the aborts rate drops by about 88% where all read-only transactions are committed.

6.5 Approximately Opaque Multi-version STM

After we applied the concept of approximated opacity on the single version STM, we applied the same concept on multi-version STM. In multi-version transactional memory each shared object may have multiple versions. In this way, when an update transaction commits, it creates new versions for the objects in its write set. Using multiple memory object versions, read-only transactions can successfully commit (without aborting at all) by preserving the versions in the read sets of the transactions. Therefore, we have modified the structure of the read/write and count object to satisfy multi-version objects.

FIGURE 6.4 (a) shows the multi-version read/write object data structure where the object x has version list ($x.vl$). Each version in $x.vl$ has the same structure of the object that is explained in FIGURE 6.1 (a). The versions in the $x.vl$ preserve the timestamp order. Actually, we add a new version to $x.vl$ every K commits

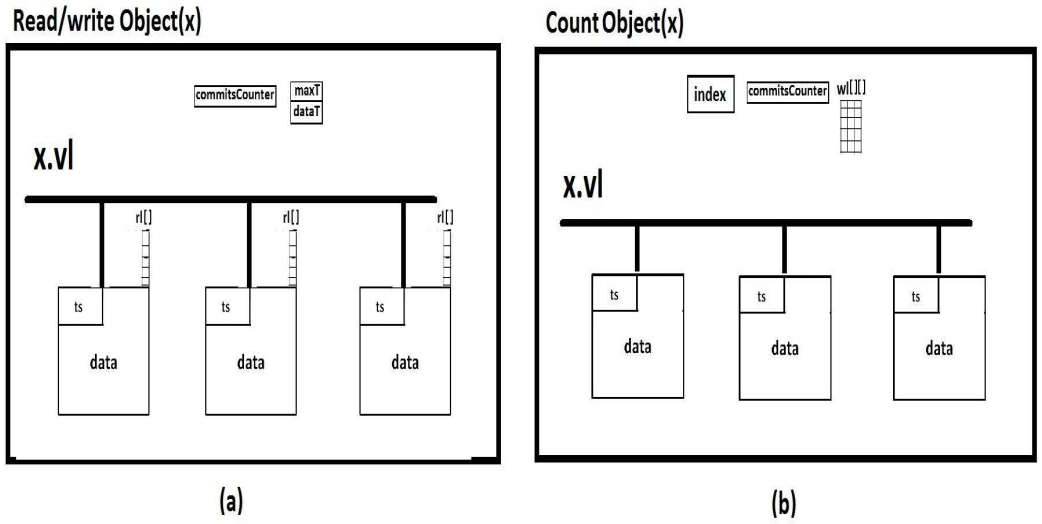

FIGURE 6.4. (a) Multi-version Read/write Object Data Structure. (b) Multi-version Count Object Data Structure.

on x. FIGURE 6.4 (b) shows the multi-version count object data structure where the count object x has version list ($x.vl$). Also each version in $x.vl$ has the same structure of the object that is explained in FIGURE 6.1 (b). Similar to multi-version read/write object the versions in the count object preserve the timestamp order and we add new version to $x.vl$ every K commits on x.

Using multi-version, we make slight modification to the algorithm, such that when the K^{th} transaction commits on x it does not overwrite the object. Instead, it creates new version and adds it to the $x.vl$. The new version has the maximum timestamp of all committed transactions which is definitely higher than the timestamps of all previous versions.

Also, in functions 7 and 8, when a read-only transaction T_i tries to read any object, if a newer transaction commits and overwrites the object, T_i aborts. However using multi-version object, if a newer transaction commits, it creates a new version of the object. Thus, T_i does not have to abort. Instead, T_i traverses $x.vl$ and finds the version with the maximum timestamp that is smaller than i. This way, the read-only transactions usually can find the suitable version in $x.vl$. The

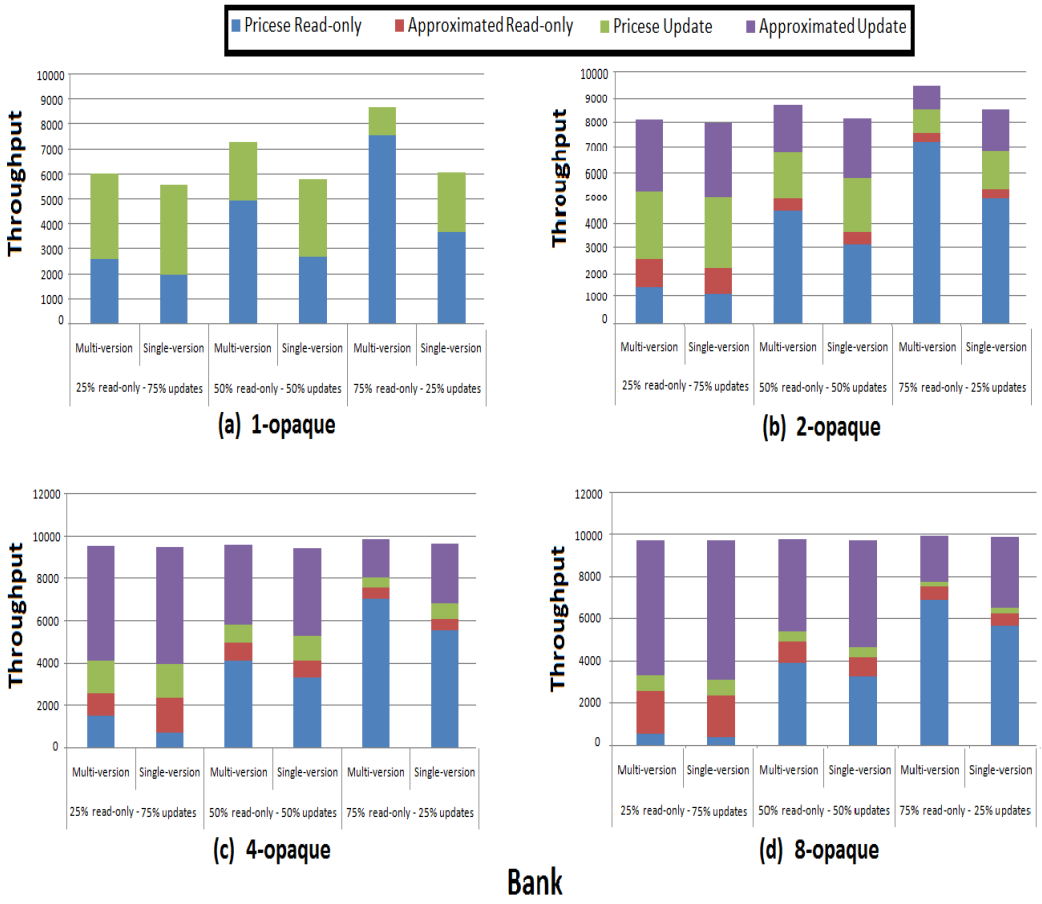

FIGURE 6.5. (a) Compare the Throughput of 1-opaque Single-version STM to 1-opaque Multi-version STM Using the Bank Benchmark. (b) Compare the Throughput of 2-opaque Single-version STM to 2-opaque Multi-version STM Using the Bank Benchmark. (c) Compare the Throughput of 4-opaque Single-version STM to 4-opaque Multi-version STM Using Bank Benchmark. (d) Compare the Throughput of 8-opaque Single-version STM to 8-opaque Multi-version STM Using the Bank Benchmark.

suitable version is the one that enables the read and counting operations to satisfy the K-legality.

6.6 Multi-version STM Experimental Results

Now we compare the performance of the single-version STM to the multi-version STM using the Bank benchmark. We use different ratios of read-only and update transactions. We run our experiments with 25% read-only and 75% update, 50% read-only and 50% update, and 75% read-only and 25% update transactions.

FIGURE 6.5 (a) Compares the throughput of 1-opaque single-version STM to 1-opaque multi-version STM using the Bank benchmark. With all different ratios of read-only and update transactions, multi-version STM usually shows better performance than the single-version one. More improvement of throughput happens when there is a higher ratio of read-only transactions, since in multi-version STM read-only transactions never abort. The same thing happens with FIGURE 6.5 (b) which compares the throughput of 2-opaque single-version STM to 1-opaque multi-version STM using the Bank benchmark. However, because of the relaxation of opacity we have some approximated read-only and approximated update transactions. In addition FIGURE 6.5 (c) and (d) show the 4-opaque and the 8-opaque executions. In both, the throughput of multi-version STM and the throughput of single-version STM are almost the same with all different ratios of read-only and update transactions. This happens because more commits to the read-only transactions results in more aborts to the update transactions. However, using multi-version STM, the ratio of precise read-only transactions is usually higher.

Chapter 7

Approximately Opaque Transactional Memory for Queue Objects

7.1 Introduction

Based on the promising results of applying approximated opacity on read-only transactions and update transactions that execute read/write and counting operations, in this chapter we apply the concept of K-opacity on queue objects, which are common objects used in typical concurrent programs. This chapter shows the design of the K-opaque algorithm for enqueue and dequeue operations, and the proof for correctness. Then, we present some experimental results to demonstrate the benefits of consistency relaxation on performance.

7.2 The Design of the Algorithm

Let us start with the structure of the TM queue object. FIGURE 7.1 shows the structure of the queue object x which has fields ts which is the maximum timestamp of the transactions that update x. $queue[][]$ is an array of two columns. The first one is for *data* and the other is for the timestamp of the transaction that enqueues or dequeues that *data*. Also x has *head* and *tail*. To show the number of concurrent transactions that perform enqueues and dequeues, we use *enqCounter* and *deqCounter*. Moreover, every queue has $wl[][]$ (writer list), which is an array of size n, where n is the number of threads in the system. In fact $wl[][]$ has three columns such that each update transaction writes to x must post its timestamp in the first column, the second column illustrates the kinds of the operation, which is either enqueue or dequeue, and the status of the transaction (which is either $live, committed$ or *aborted*) appears in the third column.

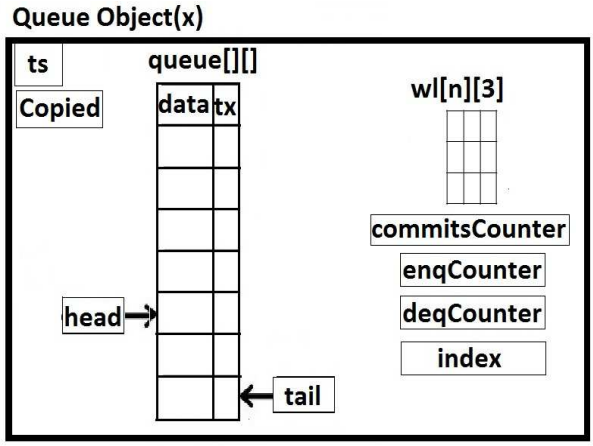

FIGURE 7.1. Queue Object Data Structure.

To start with our algorithm, the class Queue shows the structure of the objects. Basically our algorithm is timestamp-based that executes transactions concurrently but it does not update the *head* and/or *tail* objects with each commits. Indeed, every K commits on the object, a transaction overwrites it with a new value.

Main() (Algorithm 13) shows the structure of the transaction where it has a unique timestamp i using *getAndIncerement()* method that increments the value of a special object *timestamp* by 1. Transaction has a status $T_i.status$ that is set to *live* and adds itself to the *liveT* list which is used for the garbage collector in chapter 8. It also has an access set $T_i.accSet$ that is maintained during the execution. Moreover we have also two global variables which are *correct* (which is used to calculate the correct value of update transactions) and *commits* (which counts the number of commits on x).

Then, transaction T_i executes all its operations which are either enqueue and dequeue. For all operations we add the object x to $T_i.accSet$ and then execute the operation by calling function Enqueue() or Dequeue(). If the functions return $false$, then we change *valid* flag to 1 and T_i aborts immediately. When T_i executes all operations it calls TryC() and based on that it commits or abort.

In Enqueue() (Algorithm 14) if there is no version of object x in T_i's local memory, then T_i accesses the (global) shared object x, and T_i aborts if $x.ts > i$. Otherwise, if $x.ts < i$ we get $x.tail$ and we use $enqs$ which shows how many concurrent enqueues there are, and that is used to map T_i to the right spot in the $x.queue$. If the queue is full, we return nil (special specification of the queue object), or T_i enqueues the element $data$ in any position between $tail+1$ and $tail+K$. Also, we write T_i timestamp next to the element. T_i posts $i, T_i.status$ and op (enqueue or dequeue) in $wl[][]$, and returns $true$. Now we check if $x.commitsCounter = K-1$ which means that this operation is the K^{th} operation on x, then we change $x.copied$ flag to 1 and we copy x to T_i local memory.

However, if the operation finds x in T_i local memory (which means T_i already access x and executes the K^{th} operation on x) then T_i accesses the local copy. After that it will do the same procedures that would be done on the global x but on the local copy. If T_i executes another K^{th} operation on x, it just overwrites the local copy and does not touch global x.

In Dequeue() (Algorithm 15) if there is no version of object x in T_i's local memory, then T_i accesses the (global) shared object x and T_i aborts if $x.ts > i$. Otherwise, if $x.ts < i$ we get $x.tail$ and $x.head$. If the queue is empty, we return $true$ (special specification of the queue object), or we use $deqs$ (which shows the number of concurrent dequeues) to map T_i to a specific position in the queue and copy the element. T_i can be mapped to dequeue an element between $head$, and $head + K$. Also, T_i writes its information in $wl[][]$ and returns $true$. Indeed, a dequeue operation does not remove the element from the queue since T_i may abort later. Now we check if $x.commitsCounter = K-1$ which means that this operation is the K^{th} operation on x; we do the same thing in Enqueue().

In TryC() we call function CheckStatus() to know if T_i is aborted and removed from $wl[][]$ by another transaction, then we return $false$ and call function Abort(). In addition, in TryC() if another transaction copies x to its local memory (which means another transaction is committing the K^{th} operation on x), or $x.wl[][]$ has a transaction with a timestamp greater than i, then T_i aborts.

Otherwise the validation goes well and then in Commit() we check if the number of committed transactions including (T_i) equals to K. Then we abort all the other *live* transactions in $x.wl[][]$ and there are two scenarios:

i. If a copy of x is in T_i local memory, we just copy the local copy of x to global.

ii. If there is no copy of x in T_i local memory, then T_i overwrites shared object x. We adjust the $x.queue$ by deleting the elements of committed dequeues and removing the empty spots that result from aborted enqueues. In addition we update $x.ts$ and reset $x.enqCounter, x.deqCounter$ and $x.wl[][]$.

On the other hand, if the number of committed transactions does not equal to K, then we do nothing. We release all locks and we change $T_i.status$ to *committed*.

In Abort() (Algorithm 18), we change $T_i.status$ to *aborted*, and we change T_i status to aborted in the writer lists of all objects in $T_i.accSet$. Also, for any x that has a copy in T_i local memory, we reset $x.copied$ to 0.

7.3 Correctness of the Algorithm

In the correctness analysis we prove that our algorithm is K-opaque for all transactions. Let H be an arbitrary history of an execution. Let H' be a complete history that we obtain such that if a pending transaction in H didn't invoke either Commit() or Abort(), then its status is aborted, while in any other case the status is either committed or aborted, according to which of the two functions the

```
Class Queue
int ts ← 0;
int head ← 0;
int tail ← 0;
int queue[size][];
int commitsCounter ← 0;
int x.copied = 0;
//wl[][] is an array to record update transactions
//n is the number of the thread in the system
int wl[n][3];
int index ← 0;//wl[][] array's index
enqCounter ← −1;
deqCounter ← −1;
int rl[];
```

transaction invoked. Let S be the sequential execution which is a timestamp-based serialization of the transactions in H'.

Lemma 7.1. *S preserves the real time order of the transactions in H'.*

Proof. According to Main(), each transaction T_i obtains a unique timestamp using $i \leftarrow timestamp$.getAndInc(), which is an atomic operation. If $T_i <_{H'} T_j$, then $i < j$. Since S orders transactions based on their timestamp, we get $T_i <_S T_j$. In other words, $<_{H'} \subseteq <_S$, as needed. □

We continue to prove that S is K-legal with respect to any object x for any transaction.

Lemma 7.2. *The history S is K-legal, for any queue object x.*

Proof. According to the implementation of algorithms Enqueue() and Dequeue(), within a transaction T_i each individual enqueue or dequeue operation is considered in the legality specification of the queue object x. So assume that in transaction T_i, each enqueue and dequeue operation on object x is denoted as qO_i.

Algorithm 13: Main ()

```
timestamp ← 0;
int correct ← 0;//To calculate the correct value
int commits ← 0;
bool valid = true;

Upon receipt of a transaction do;

foreach (transaction T) do
    //Get a unique timestamp
    i ← Timestamp.getAndInc();
    T_i.status ← live;
    /* liveT is used for garbage collector */
    liveT ← i ∪ liveT;
    T_i.accSet ← ∅;
    while (there is unexecuted operation on object x) do
        switch operation on x do
            case (x.enqueue(data))
                //Enqueue operation
                enqs ← x.enqCounter.getAndInc();//How many concurrent
                enqueues
                T_i.accSet ← x ∪ T_i.accSet;
                valid = enqueue(i, x, data, enqs);
            case (x.dequeue(data))
                //Dequeue operation
                deqs ← x.deqCounter.getAndInc();//How many concurrent
                dequeues
                T_i.accSet ← x ∪ T_i.accSet;
                valid = dequeue(i, x, deqs);
        if (valid = false) then
            Abort(i, T_i.accSet);
            return;

    //Based on TryC(), transaction commits or aborts
    if (TryC(i, T_i.accSet)) then
        Commit(i, T_i.accSet);
    else
        Abort(i, T_i.accSet);
    return;
```

Algorithm 14: Enqueue$(i, x, data, enqs)$

```
//If x is not in any transaction's local memory
if (x.copied = 0) then
    //If a newer transaction overwrites x, or there are more than K dequeues,
    then T_i aborts
    if ((x.ts > i) ∨ (enqs >= K)) then
        return false;
    t ← x.tail;
    //We check if the queue is full then we cannot map it
    if (t = size) then
        //The queue is full
        return nil;
    //Mapping
    position ← enqs + t + 1;
    //Insert element
    x.queue[position].data ← data;
    x.queue[position].tx ← i;
    x.wl[index].tx ← i;
    x.wl[index].op ← enqueue;
    x.wl[index].status ← live;
    //Now we check if it is the K^th operation
    if (x.commitsCounter = K − 1) then
        x.copied = i;
        //We prepar a local version of x
        copy x to T_i local memory;
        adjust x_local.queue[][];
        max = max(x_local.queue[].tx);//Maximum timestamp in x_local.queue[][]
        update x_local.head; update x_local.tail; x_local.ts ← max;
        reset x_local.enqCounter; x.localdeqCounter; x_local.wl[][];
        x_local.commitsCounter = 0;

else
    //If x is in another transaction local memory
    if (x.copied ≠ i) then
        return false;
    do the same thing (in if() part) on the local copy but we do not copy x to
    local memory again;
return true;
```

Algorithm 15: Dequeue$(i, x, deqs)$

```
//If x is not in any transaction's local memory
if (x.copied = 0) then
    //If a newer transaction overwrites x, or there are more than K dequeues,
    then T_i aborts
    if ((x.ts > i) ∨ (deqs >= K)) then
        return false;
    h ← x.head;
    t ← x.tail;
    //We check if the queue is empty then we cannot map it
    if ((t = 0) ∨ ((t − h) + 1) < deqs)) then
        //Empty queue
        return nil;
    //Mapping
    position ← deqs + (h + 1);
    //Get element
    data ← x.queue[position].data;
    x.queue[position].tx ← i;
    x.wl[index].tx ← i;
    x.wl[index].op ← dequeue;
    x.wl[index].status ← live;
    //Now we check if it is the K^th operation
    if (x.commitsCounter = K − 1) then
        x.copied = 1;
        //We prepar a local version of x
        copy x to T_i local memory;
        adjust x_local.queue[][];
        max = max(x_local.queue[].tx);//Maximum timestamp in
        xlocal.queue[][]
        update x_local.head; update x_local.tail; x_local.ts ← max;
        reset x_local.enqCounter; x._localdeqCounter; x_local.wl[][];
        x_local.commitsCounter = 0;

else
    //If x is in another transaction local memory
    if (x.copied ≠ i) then
        return false;
    do the same thing (in if() part) on the local copy but we do not copy x to
    local memory again;
return true;
```

Algorithm 16: CheckStatus(i, x)

```
int T_iRemoved ← 0;
//Check if T_i is aborted and removed from wl by another transaction
for ( from m ← 0 to m = n) do
    if (x.wl[m].tx = i) then
        T_iRemoved ← 1;//It is not removed
        break;
//If T_i is removed from wl
if (T_iRemoved = 0) then
    commits = −1;
    return commits;
//T_i still in wl
x.wl[m].status = committed;
//Check how many committed transactions
commits = x.commitsCounter;
return commits;
```

Algorithm 17: TryC$(i, T_i.accSet)$

```
L ← ∅;
forall the (x in T_i.accSet) do
    lock();
    L ← L ∪ x;
    commits ←CheckStatus(i, x);
    if ((commits = −1) ∨ (x.wl[][] has a transaction T_m where m > i)) then
        Unlock() all objects in L;
        return false;
return true;
```

Algorithm 18: Abort$(i, T_i.accSet)$

```
T_i.status ← aborted;
//Change its status in all counters and queues it accesses
forall the (x in T_i.accSet) do
    if (x.copied = i) then
        x.copied = 0;
    x.wl[x.index].status ← aborted;
    remove i from x.rl[];
return;
```

Algorithm 19: Commit($i, T_i.accSet$)

```
int max ← −1;
T_i.status ← committed;
forall the (x in T_i.accSet) do
    x.commitsCounter + +;
    if (a version of x ∈ T_i.accSet) then
        abort the other live transactions in x.wl[][];
        let x = x_local;
    else if (x.commitsCounter = K) then
        abort the other live transactions in x.wl[][];
        adjust x.queue[][];
        max = max(queue[].tx); //Maximum timestamp in queue[][]
        update x.head; update x.tail; x.ts ← max;
        reset x.enqCounter; x.deqCounter; x.wl[][];
        reset x.copied; x.commitsCounter;
    else
        //Just Commit
        x.wl[].status = Committed;
Unlock() all objects in L;
return true;
```

Let $S(x) = T_1, T_2, \ldots, T_q$ be the subsequence of transactions in S that invoke enqueue or dequeue operations to x. A transaction T_g that overwrites x is a transaction that commits and updates $x.queue[][]$, $x.head$ and $x.tail$. It also sets $x.copied = 0$, $x.ts = g$ and $x.commitsCounter = 0$; Let $S'(x)$ denote the subsequence of $S(x)$ consisting of all transactions that commit. Let $S''(x)$ denote the subsequence of $S(x)$ consisting of all transactions that overwrite x. Let $V(x)$ denote all individual operations in $S(x)$ and let $V'(x)$ denote all individual operations in $S'(x)$ while $V''(x)$ denote all individual operations in $S''(x)$. For any queue operation $qOp_{g,u}$, g is the timestamp of the transaction that executes $qOp_{g,u}$ and u is the order of $qOp_{g,u}$ in V.

Let qs denote the queue status and $qs_{0,0}, \ldots, qs_{q,d}$ be the respective sequence of returned queue status of the enqueue and dequeue operations of transactions

in $S(x)$, namely, $qs_{g,first}, \ldots, qs_{g,last}$ represent the queue statuses returned by T_g. Let $sq'_{g,u}$ denote the *precise queue status* that would have been returned by any queue operation of transaction T_g when the consistency is precise, that is, $qs'_{g,u} = qOp_{0,0} \wedge \sum_{(1 \leq l < j) \wedge qOp_{l,s} \in V'(x)} qOp_{l,s}$, where $qOp_{0,0}$ is the initial value of x. Let $o_{g,u} = qs_{g,a} \wedge qOp_{g,u}$ be the result of any queue operation of transaction T_g. Similarly, let $o'_{g,u} = qs'_{g,u} \wedge qOp_{g,u}$ be the respective *precise* result of the queue operation of transaction T_g.

For each $qOp_{g,u} \in V(x)$ let $P_{g,u}$ denote the set of the last K operations which precede $qOp_{g,u}$ in $V(x)$. We only need to prove the following two properties:

i. For each $qOp_{g,u} \in V''(x)$, $qs_{g,u} = qs'_{g,u}$ (the operations in $V''(x)$ are precise.

ii. For each $qOp_{g,u} \in V(x) \setminus V''(x)$, either $qs_{g,u} = o_{r,z}$ and $qOp_{r,z} \in P_{g,u}$ and $qOp_{r,z} \in V''(x)$, or $qs_{g,u} = qs_{0,0}$ and $|P_{g,u}| < K$.

We prove these properties by induction on q. For $q = 0$, $V(x)$ is empty and the properties hold trivially. Assume now that the properties hold for any $q < n$; we will show that the properties hold also for $q = n > 0$.

Consider now the last operation $qOp_{i,n}$. According to function Enqueue() and Dequeue(), $qOp_{i,n}$ checks $x.ts$ that shows the timestamp of the last transaction that overwrites x. Suppose that $x.ts = j \geq i$. Let Q be the set of operations that appear in V between $qOp_{j,a} \in T_j$ and $qOp_{i,n} \in T_i$ and that have queue operations to x and commit (Q does not contain $qOp_{j,a}$ or $qOp_{i,n}$). We continue to show that $|Q| \leq K - 1$.

We first show that none of the operations in Q overwrite x. Suppose for the sake of contradiction that there is an operation $qOp_{m,b} \in Q$ which overwrites x. Note that $j, a \leq m, b \leq i$. We examine three cases with respect to when $qOp_{m,b}$ belongs to a committed transaction T_m in H':

- T_m commits before T_j commits.

  In this case, when T_j invokes TryC() it observes one of the following scenarios:

  - $m \in x.wl$ and T_j observes that a transaction T_m with higher timestamp ($m > j$) has committed on x but it does not overwrite x, and hence T_j aborts (considering that $j \neq m \neq i$).

  - $x.ts \geq j$: Since T_m commits and overwrites x based on the Algorithm Commit(), T_m aborts all live transactions with smaller timestamp than m, and hence T_j aborts (considering that $j \neq m \neq i$).

  - Since some of these operations may belong to the same transaction, and since T_m commits before T_j commits, then $qOp_{j,a}$ and $qOp_{m,b}$ belong to different transactions.

    1. Now let us assume that $j = i$ (which means $qOp_{j,a}$ and $qOp_{i,n}$ belong to the same transaction). Based on Enqueue() and Dequeue(), $qOp_{j,a}$ would copy x to T_j local memory and $qOp_{i,n}$ would read from the local copy. Then $qOp_{m,b}$ would find that the object is copied by another concurrent transaction, so T_m aborts and $qOp_{m,b}$ would not execute between $qOp_{j,a}$ and $qOp_{i,n}$ which means it cannot be in Q, contradiction.

    2. If $m = i$ and then $qOp_{m,b}$ would copy x to T_m local memory and $qOp_{i,n}$ would read the from local copy, which contradicts with our assumption that $qOp_{i,n}$ reads $qOp_{j,a}$.

  In all scenarios, either T_j aborts, which is impossible, or $qOp_{m,b}$ is not in Q.

- T_m commits and overwrites x before $qOp_{i,n}$ starts.

  In this case, $qOp_{i,n}$ reads either the value of $x.head$ and/or $x.tail$ that is written by $qOp_{m,b}$ or by a more recent operation. However, this contradicts the assumption that $qOp_{i,n}$ reads $qOp_{j,n}$. Actually, this also holds if $j = m$, $j = m = i$ or $m = i$.

  On the other hand, if $j = i$, then $qOp_{j,a}$ would copy x to T_j local memory and $qOp_{m,b}$ would find that x is copied. So T_m aborts and it cannot execute between $qOp_{j,a}$ and $qOp_{i,n}$. Thus, $qOp_{m,b} \notin Q$, contradiction.

- T_m commits and overwrites x after $qOp_{i,n}()$ ends.

  In this case, we have the following scenarios:

  - In the case of $j \neq m \neq i$, in its TryC() transaction T_m will observe that $x.commitsCounter \geq K - 1$ (which means T_m is the K^{th} transaction), and also it observes that T_i is in the writer list of x (that is, $i \in x.wl$ with $i > m$), and the combination of these two observations together force T_m to abort, which is a contradiction.
  - Since some of these operations may belong to the same transaction, it observes one of the following scenarios:

    1. Since T_m commits and overwrites x after $qOp_{i,n}()$ ends, then we cannot have that $m = i$ or $j = m = i$; otherwise, $qOp_{m_b} \notin Q$

    2. If $j = i$, then x would be copied by T_j and $qOp_{m,b} \notin Q$.
    3. If $j = m$, then x would be copied by T_j, and T_i would abort and $qOpi, n$ cannot execute.

Therefore, no operation $qOp_{m,b}$ in Q overwrites x. This implies that each operation in Q increments $x.commitsCounter$. Therefore, similar to the proof of Lemma 6.2 the operations in Q assign unique values to $x.commitsCounter$. Since the $x.commitsCounter$ cannot exceed $K-1$ and none of the operations in Q set $x.commitsCounter = 0$, we get that $|Q| \leq K-1$. The same observations for Q hold even if $qOp_{i,n}$ reads the initial value of x and $qOp_{j,a}$ is replaced by a special event that initializes x.

Suppose now that $qOp_{i,n} \in V''$, which implies that $T_i \in S''$. When T_i invokes Algorithm Commit(), the only committed transactions in the writer list of x are the ones in set Q. Therefore, the value returned by the $qOp_{i,n}$ of T_i is equal to $qs_{i,n} = qs_j \wedge \sum_{qOp_l \in Q} qOp_{i_n}$. By induction hypothesis, $qs_{j,a} = qs'_{j,a}$, and hence $qs_{i,n} = qs'_{i,n}$, and therefore, property (i) holds. If $qOp_{i,n} \in V \setminus V''$ (which implies $T_i \in S \setminus S''$), then it returns $qs_{i,n} = qs_{j,a} = qs'_{j,a}$. Since, $Q \cup \{qOp_{j,a}\} \subseteq P_{i,n}$ property (ii) holds as well. (Note that properties (i) and (ii) hold even if $qOp_{i,n}$ reads the initial value $qs_{0,0}$ and $qOp_{j,a}$ is replaced with a special initialization event of x.) □

Based on Lemmas 7.1 and 7.2 we obtain the following theorem.

Theorem 7.3. *Any execution history H of our algorithm is K-opaque.*

7.4 Experimental Results

In our experiment, we simulate Linked-list benchmarks from TinySTM-1.0.5 [13], but we modify the structure of the object and the algorithm to match our specifications. We run the experiments on a machine with dual Intel(R) Xeon(R) CPU E5-2630 (6 cores total) clocked at 2.30 GHz. Each run of the benchmark takes about 7000 milliseconds using 10 threads. We test the benchmark to compare the opaque execution (1-opaque) with 2-opaque, 4-opaque and 8-opaque. We use the

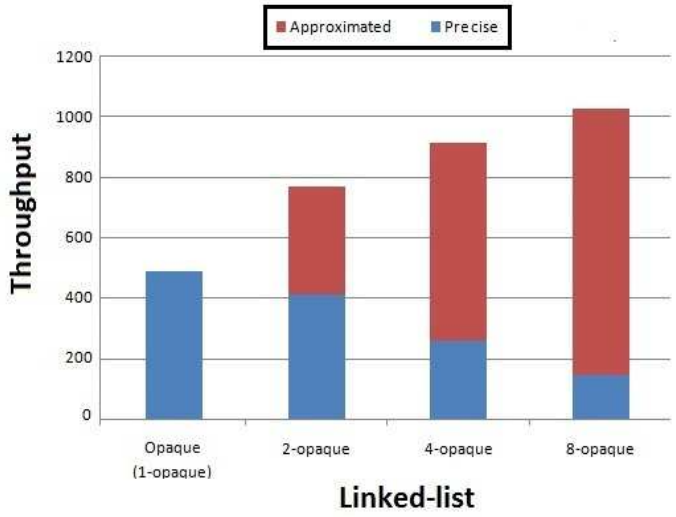

FIGURE 7.2. Comparison of the Throughput of 1-opaque, 2-opaque, 4-opaque and 8-opaque Executions on the Linked-list Benchmark.

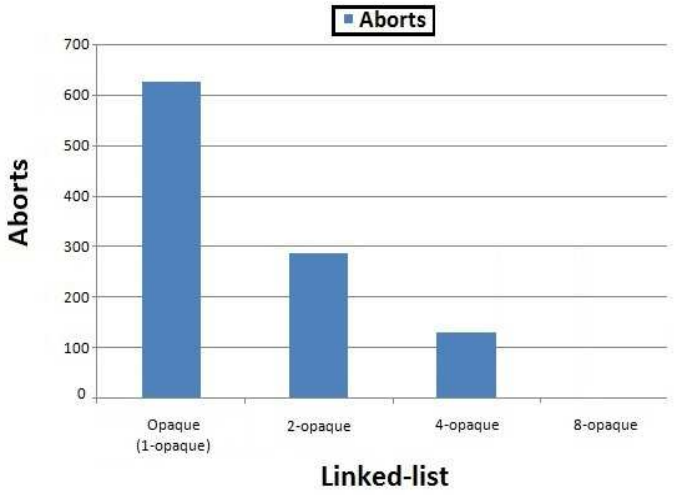

FIGURE 7.3. Comparison of the Aborts Rate of 1-opaque, 2-opaque, 4-opaque and 8-opaque Executions on the Linked-list Benchmark.

Linked-list benchmark to test the queue operations. In the Linked-list we initialize 70 lists (where we modify the structure of each list to be a queue object) and we have that *add* and *delete* nodes to simulate enqueue and dequeue operations. Both enqueue and dequeue operations are considered as update transactions. In our execution we generate 50% enqueue operations and 50% dequeue operations.

FIGURE 7.2 demonstrates the throughput of 1-opaque, 2-opaque, 4-opaque and 8-opaque executions on the Linked-list benchmark.

The throughput of the 8-opaque execution is the maximum since the throughput improves as we relax the opacity. Moreover FIGURE 7.3 shows the aborts rate of 1-opaque, 2-opaque, 4-opaque and 8-opaque executions on the Linked-list benchmark. The figure illustrates the drop of the aborts rate as we relax opacity.

Chapter 8
Garbage Collector

8.1 Introduction

This chapter is previously published by IEEE [1]. The basic idea behind having multiple versions of each object is to avoid some aborts and guarantee progressiveness. However, there is a negative effect due to the increase on the space requirements. So, many multi-version STMs have garbage collection procedure to delete old versions which are no longer needed by active transactions. Some multi-version STMs keep a specific number of versions for each object, such as 4 or 8 versions [29]. However, this may not be very efficient as the determined number of versions may be smaller than what is needed for some objects or larger than what is needed for others. In this chapter we introduce our garbage collector GB to decide which versions are wanted or unwanted for each object dynamically. Now we present the design and the algorithm of GB. Then we show the correctness and efficiency proofs.

8.2 Design of the GB

Our garbage collector (Algorithm 20) decides which versions are wanted or unwanted for each object dynamically. It does not remove the last written version of an object, but it just works on the saved version list $x.vl$ for all objects x. To find unneeded versions for an object x, the garbage collector finds the minimum timestamp of the live transactions $minliveT$. In fact, we record the live transactions in a list that is called $liveT$. Then, it finds the version with the largest timestamp smaller than $minliveT$, and this version is denoted as $maxvl$. Now, the garbage

[1] This chapter previously appeared as [Basem Assiri and Costas Busch, Approximately Opaque Multi-version Permissive Transactional Memory, published by The Institute of Electrical and Electronics Engineers (IEEE)]. See the permission letter in Appendix.]

collector keeps any version with timestamp greater than or equal to $maxvl$ and deletes any version with timestamp smaller than it.

Algorithm 20: Garbage Collector

```
minliveT ← smallest timestamp of any transaction in liveT;
foreach object x do
    maxvl ← 0;
    forall the (ts, data, rl) ∈ x.vl do
        if ts > maxvl ∧ ts < minliveT then
            maxvl ← ts;
    forall the (ts, data, rl) ∈ x.vl do
        if ts < maxvl then
            /* delete (ts, data, rl) from x.vl */
            x.vl ← x.vl \ {(ts, data, rl)};
```

8.3 Correctness of the Algorithm

Theorem 8.1. *Our garbage collector Algorithm 20 does not violate the correctness of our execution.*

Proof. Our garbage collector works only on the saved version list of each object, $x.vl$. It finds the minimum timestamp of the live transactions $minliveT$ and the version $maxvl$ which has the largest timestamp smaller than $minliveT$. Then, it deletes all saved versions with timestamps smaller than $maxvl$.

Let $v \in x.vl$ be the version of x with $v.ts = maxvl$. Assume for contradiction that our garbage collector deletes a needed saved version of object x, say version v', with $v'.ts < maxvl$. Note that the version v' must be created and stored in $x.vl$ before v, since the intervals of the respective transactions that create v and v' would overlap, and hence one of these transaction would have to abort according to function Validate().

Let T_i be the live transaction that needs v'. We know that $i \geq minliveT$, and hence $i > maxvl$. Transaction T_i has to be read-only, since it accesses v' which is

stale (update transactions only access the $x.lastCommit$ version which must have timestamp at least $maxvl$). Therefore, when transaction T_i invokes the function GetLatestVersion() it has to read v and not v', since $v'.ts < v.ts < i$; a contradiction. □

Assume that the transactions in our algorithm access the set of objects $O = (x_1, x_2, ..., x_n)$. Let V be the set of all committed versions and V' the set of all saved versions.

Theorem 8.2. *There are execution scenarios in which our garbage collector can reduce the space required to no more than* $2|O|$ *saved versions.*

Proof. The worst case scenario for space requirement happens when for all objects on the system, there is at least one pending transaction on each version in object vl. So, all versions in the system are still needed and the garbage collector cannot delete any version. In this case, the garbage collector space complexity remains the same. Based on Theorem 5.7, the worst case scenario space complexity (max size of $|V'|$) of our algorithm without the garbage collector will be $\Theta(|V|/K + |O|)$.

However, in all other cases the garbage collector can delete some versions and improve the space complexity. Suppose the best case scenario is when there are some pending transaction working only on the last written versions in some objects' saved version list vl, or when there are no live transactions at all. In this case, the space complexity of our algorithm without the garbage collector according to Theorem 5.7 is $\Theta(|V|/K + |O|)$. While using the garbage collector, the needed versions for each object are two, which are the $lastCommit$ and the last saved version $maxvl$. So, the space complexity drops to $2|O|$. □

Chapter 9
Conclusion

In conclusion, our scheduling algorithm uses the idea of supervised and unsupervised machine learning models to improve the performance of transactional memory algorithms. This way of scheduling allows more flexibility to find the suitable scheduling for different problems. Indeed finding the suitable scheduling FV helps to increase the throughput and to decrease the aborts. Our results show that the types and the durations of transactions extremely impact the performance of transactions execution. In addition, the frequent calls of learning model allows for the to change the scheduling ratio in response to any changes of arriving transactions. In the future, we want to test some other learning techniques on different algorithms.

Furthermore, we have introduced the notion of K-opacity, which is useful in systems which allow some transactions to return approximated values. A benefit of K-opacity is that it enables the system to save a smaller number of object versions. We show that more relaxation to the opacity helps to increase the throughput and reduce aborts. Our algorithm guarantees progressiveness such that it never aborts transactions unless if it is necessary for the correctness of the execution. Moreover, the algorithm never aborts read-only transactions, and at least one of the conflicted update transactions can commit.

In addition, count object can be extended to execute other arithmetic operations such as multiplication and division. Therefore, we can add another column in the wl array to record the type of the arithmetic operations. The order of operations (or operator precedence) will be based on the transactions timestamps.

Also the timestamp order of the sequential history S is used to prove the correctness of transactions execution, while it is possible to find another S that does not respect the timestamps order but it is more flexible to allow more commits.

Moreover, the concept of K-opacity works with transactions that access read/write and count objects as well as transactions that access queue objects. However, obviously the approximated opacity concept is composable such that we may have a transaction that accesses read/write, count and queue objects. For future works, the composability of approximated opacity can be extended to be applied on other data structures such as stack and linked list.

References

[1] Swarup Acharya, Phillip B Gibbons, and Viswanath Poosala. Aqua: A fast decision support systems using approximate query answers. In *Proceedings of the 25th International Conference on Very Large Data Bases*, pages 754–757. Morgan Kaufmann Publishers Inc., 1999.

[2] George S Almasi and Allan Gottlieb. *Highly parallel computing*. Benjamin-Cummings Publishing Co., Inc., 1989.

[3] Naomi S Altman. An introduction to kernel and nearest-neighbor nonparametric regression. *The American Statistician*, 46(3):175–185, 1992.

[4] SS Arya and S Lavanya. A comparative study of machine learning approaches-svm and ls-svm using a web search engine based application. *International Journal on Computer Science & Engineering*, 4(5), 2012.

[5] Hagit Attiya and Eshcar Hillel. Single-version stms can be multi-version permissive. In *Distributed Computing and Networking*, pages 83–94. Springer, 2011.

[6] Utku Aydonat and Tarek Abdelrahman. Serializability of transactions in software transactional memory. In *Second ACM SIGPLAN Workshop on Transactional Computing*, 2008.

[7] João Cachopo and António Rito-Silva. Versioned boxes as the basis for memory transactions. *Science of Computer Programming*, 63(2):172–185, 2006.

[8] Chen Chen, Xifeng Yan, Feida Zhu, and Jiawei Han. gapprox: Mining frequent approximate patterns from a massive network. In *Data Mining, 2007. ICDM 2007. Seventh IEEE International Conference on*, pages 445–450. IEEE, 2007.

[9] Edgar F Codd, Sharon B Codd, and Clynch T Salley. Providing olap (on-line analytical processing) to user-analysts: An it mandate. *Codd and Date*, 32, 1993.

[10] Joseph Devietti, Brandon Lucia, Luis Ceze, and Mark Oskin. Dmp: deterministic shared memory multiprocessing. In *ACM Sigplan Notices*, volume 44, pages 85–96. ACM, 2009.

[11] Pierangelo Di Sanzo, Marco Sannicandro, Bruno Ciciani, and Francesco Quaglia. On exploring markov chains for transaction scheduling optimization in transactional memory. *7th Workshop on the Theory of Transactional Memory (WTTM 2015)*.

[12] Pascal Felber, Christof Fetzer, Patrick Marlier, and Torvald Riegel. Time-based software transactional memory. *Parallel and Distributed Systems, IEEE Transactions on*, 21(12):1793–1807, 2010.

[13] 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*, pages 237–246. ACM, 2008.

[14] Rachid Guerraoui, Thomas A Henzinger, and Vasu Singh. Permissiveness in transactional memories. In *Distributed Computing*, pages 305–319. Springer, 2008.

[15] Rachid Guerraoui and Michal Kapalka. Opacity: A correctness condition for transactional memory. Technical report, 2007.

[16] Rachid Guerraoui and Michal Kapalka. On the correctness of transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 175–184. ACM, 2008.

[17] Rachid Guerraoui and Michal Kapalka. The semantics of progress in lock-based transactional memory. *ACM SIGPLAN Notices*, 44(1):404–415, 2009.

[18] Trevor Hastie, Robert Tibshirani, Jerome Friedman, and James Franklin. The elements of statistical learning: data mining, inference and prediction. *The Mathematical Intelligencer*, 27(2):83–85, 2005.

[19] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ISCA '93, pages 289–300, New York, NY, USA, 1993. ACM.

[20] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming, Revised Reprint*. Elsevier, 2012.

[21] Xuedong D Huang, Yasuo Ariki, and Mervyn A Jack. *Hidden Markov models for speech recognition*, volume 2004. Edinburgh university press Edinburgh, 1990.

[22] Stephen Cole Kleene. Introduction to metamathematics. 1952.

[23] Leonid Aryeh Kontorovich, Corinna Cortes, and Mehryar Mohri. Kernel methods for learning languages. *Theoretical Computer Science*, 405(3):223–236, 2008.

[24] Priyanka Kumar, Sathya Peri, and K. Vidyasankar. A timestamp based multi-version stm algorithm. In *Distributed Computing and Networking*, pages 212–226. Springer Berlin Heidelberg, 2014.

[25] Karl Ljungkvist, Martin Tillenius, Sverker Holmgren, Martin Karlsson, and Elisabeth Larsson. Early results using hardware transactional memory for high-performance computing applications. In *Proc. 3rd Swedish Workshop on Multi-Core Computing*, pages 93–97. Göteborg, Sweden: Chalmers University of Technology, 2010.

[26] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[27] Dmitri Perelman, Anton Byshevsky, Oleg Litmanovich, and Idit Keidar. Smv: selective multi-versioning stm. In *Distributed Computing*, pages 125–140. Springer, 2011.

[28] Dmitri Perelman, Rui Fan, and Idit Keidar. On maintaining multiple versions in stm. In *Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, pages 16–25. ACM, 2010.

[29] Torvald Riegel, Pascal Felber, and Christof Fetzer. A lazy snapshot algorithm with eager validation. In *Distributed Computing*, pages 284–298. Springer, 2006.

[30] Adrian Sampson, Jacob Nelson, Karin Strauss, and Luis Ceze. Approximate storage in solid-state memories. *ACM Transactions on Computer Systems (TOCS)*, 32(3):9, 2014.

[31] William N Scherer III and Michael L Scott. Contention management in dynamic software transactional memory. In *PODC Workshop on Concurrency and Synchronization in Java programs*, pages 70–79, 2004.

[32] William N Scherer III and Michael L Scott. Advanced contention management for dynamic software transactional memory. In *Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing*, pages 240–248. ACM, 2005.

[33] Eko Setiawan and Adharul Muttaqin. Implementation of k-nearest neightbors face recognition on low-power processor. *TELKOMNIKA (Telecommunication Computing Electronics and Control)*, 13(3), 2015.

[34] Nir Shavit and Dan Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, 1997.

[35] Konrad Siek and Paweł T Wojciechowski. Brief announcement: Relaxing opacity in pessimistic transactional memory. *DISC*, pages 540–541, 2014.

[36] Mukul K Sinha. Nonsensitive data and approximate transactions. *Software Engineering, IEEE Transactions on*, (3):314–322, 1983.

[37] Qingping Wang, Sameer Kulkarni, John Cavazos, and Michael Spear. A transactional memory with automatic performance tuning. *ACM Transactions on Architecture and Code Optimization (TACO)*, 8(4):54, 2012.

[38] Haifeng Yu and Amin Vahdat. Building replicated internet services using tact: A toolkit for tunable availability and consistency tradeoffs. In *Advanced Issues of E-Commerce and Web-Based Information Systems, 2000. WECWIS 2000. Second International Workshop on*, pages 75–84. IEEE, 2000.

Appendix. Copyrights Letters

IEEE COPYRIGHT AND CONSENT FORM

To ensure uniformity of treatment among all contributors, other forms may not be substituted for this form, nor may any wording of the form be changed. This form is intended for original material submitted to the IEEE and must accompany any such material in order to be published by the IEEE. Please read the form carefully and keep a copy for your files.

Transactional Memory Scheduling Using Machine Learning Techniques
Basem Assiri and Costas Busch
2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing

COPYRIGHT TRANSFER

The undersigned hereby assigns to The Institute of Electrical and Electronics Engineers, Incorporated (the "IEEE") all rights under copyright that may exist in and to: (a) the Work, including any revised or expanded derivative works submitted to the IEEE by the undersigned based on the Work; and (b) any associated written or multimedia components or other enhancements accompanying the Work.

GENERAL TERMS

1. The undersigned represents that he/she has the power and authority to make and execute this form.
2. The undersigned agrees to indemnify and hold harmless the IEEE from any damage or expense that may arise in the event of a breach of any of the warranties set forth above.
3. The undersigned agrees that publication with IEEE is subject to the policies and procedures of the IEEE PSPB Operations Manual.
4. In the event the above work is not accepted and published by the IEEE or is withdrawn by the author(s) before acceptance by the IEEE, the foregoing grant of rights shall become null and void and all materials embodying the Work submitted to the IEEE will be destroyed.
5. For jointly authored Works, all joint authors should sign, or one of the authors should sign as authorized agent for the others.
6. The author hereby warrants that the Work and Presentation (collectively, the "Materials") are original and that he/she is the author of the Materials. To the extent the Materials incorporate text passages, figures, data or other material from the works of others, the author has obtained any necessary permissions. Where necessary, the author has obtained all third party permissions and consents to grant the license above and has provided copies of such permissions and consents to IEEE

You have indicated that you DO wish to have video/audio recordings made of your conference presentation under terms and conditions set forth in "Consent and Release."

CONSENT AND RELEASE

1. In the event the author makes a presentation based upon the Work at a conference hosted or sponsored in whole or in part by the IEEE, the author, in consideration for his/her participation in the conference, hereby grants the IEEE the unlimited, worldwide, irrevocable permission to use, distribute, publish, license, exhibit, record, digitize, broadcast, reproduce and archive, in any format or medium, whether now known or hereafter developed: (a) his/her presentation and comments at the conference; (b) any written materials or multimedia files used in connection with his/her presentation; and (c) any recorded interviews of him/her (collectively, the "Presentation"). The permission granted includes the transcription and reproduction of the Presentation for inclusion in products sold or distributed by IEEE and live or recorded broadcast of the Presentation during or after the conference.
2. In connection with the permission granted in Section 1, the author hereby grants IEEE the unlimited, worldwide, irrevocable right to use his/her name, picture, likeness, voice and biographical information as part of the advertisement, distribution and sale of products incorporating the Work or Presentation, and releases IEEE from any claim based on right of privacy or publicity.

BY TYPING IN YOUR FULL NAME BELOW AND CLICKING THE SUBMIT BUTTON, YOU CERTIFY THAT SUCH ACTION CONSTITUTES YOUR ELECTRONIC SIGNATURE TO THIS FORM IN ACCORDANCE WITH UNITED STATES LAW, WHICH AUTHORIZES ELECTRONIC SIGNATURE BY AUTHENTICATED REQUEST FROM A USER OVER THE INTERNET AS A VALID SUBSTITUTE FOR A WRITTEN SIGNATURE.

Basem Assiri	26-11-2015
Signature	**Date (dd-mm-yyyy)**

Information for Authors

AUTHOR RESPONSIBILITIES

The IEEE distributes its technical publications throughout the world and wants to ensure that the material submitted to its publications is properly available to the readership of those publications. Authors must ensure that their Work meets the requirements as stated in section 8.2.1 of the IEEE PSPB Operations Manual, including provisions covering originality, authorship, author responsibilities and author misconduct. More information on IEEE's publishing policies may be found at http://www.ieee.org/publications_standards/publications/rights/authorrightsresponsibilities.html Authors are advised especially of IEEE PSPB Operations Manual section 8.2.1.B12: "It is the responsibility of the authors, not the IEEE, to determine whether disclosure of their material requires the prior consent of other parties and, if so, to obtain it." Authors are also advised of IEEE PSPB Operations Manual section 8.1.1B: "Statements and opinions given in work published by the IEEE are the expression of the authors."

RETAINED RIGHTS/TERMS AND CONDITIONS

- Authors/employers retain all proprietary rights in any process, procedure, or article of manufacture described in the Work.
- Authors/employers may reproduce or authorize others to reproduce the Work, material extracted verbatim from the Work, or derivative works for the author's personal use or for company use, provided that the source and the IEEE copyright notice are indicated, the copies are not used in any way that implies IEEE endorsement of a product or service of any employer, and the copies themselves are not offered for sale.
- Although authors are permitted to re-use all or portions of the Work in other works, this does not include granting third-party requests for reprinting, republishing, or other types of re-use.The IEEE Intellectual Property Rights office must handle all such third-party requests.
- Authors whose work was performed under a grant from a government funding agency are free to fulfill any deposit mandates from that funding agency.

AUTHOR ONLINE USE

- **Personal Servers**. Authors and/or their employers shall have the right to post the accepted version of IEEE-copyrighted articles on their own personal servers or the servers of their institutions or employers without permission from IEEE, provided that the posted version includes a prominently displayed IEEE copyright notice and, when published, a full citation to the original IEEE publication, including a link to the article abstract in IEEE Xplore. Authors shall not post the final, published versions of their papers.
- **Classroom or Internal Training Use.** An author is expressly permitted to post any portion of the accepted version of his/her own IEEE-copyrighted articles on the author's personal web site or the servers of the author's institution or company in connection with the author's teaching, training, or work responsibilities, provided that the appropriate copyright, credit, and reuse notices appear prominently with the posted material. Examples of permitted uses are lecture materials, course packs, e-reserves, conference presentations, or in-house training courses.
- **Electronic Preprints.** Before submitting an article to an IEEE publication, authors frequently post their manuscripts to their own web site, their employer's site, or to another server that invites constructive comment from colleagues. Upon submission of an article to IEEE, an author is required to transfer copyright in the article to IEEE, and the author must update any previously posted version of the article with a prominently displayed IEEE copyright notice. Upon publication of an article by the IEEE, the author must replace any previously posted electronic versions of the article with either (1) the full citation to the

IEEE COPYRIGHT AND CONSENT FORM

To ensure uniformity of treatment among all contributors, other forms may not be substituted for this form, nor may any wording of the form be changed. This form is intended for original material submitted to the IEEE and must accompany any such material in order to be published by the IEEE. Please read the form carefully and keep a copy for your files.

Approximately Opaque Multi-version Permissive Transactional Memory
Basem Assiri and Costas Busch
2016 45th International Conference on Parallel Processing Workshops

COPYRIGHT TRANSFER

The undersigned hereby assigns to The Institute of Electrical and Electronics Engineers, Incorporated (the "IEEE") all rights under copyright that may exist in and to: (a) the Work, including any revised or expanded derivative works submitted to the IEEE by the undersigned based on the Work; and (b) any associated written or multimedia components or other enhancements accompanying the Work.

GENERAL TERMS

1. The undersigned represents that he/she has the power and authority to make and execute this form.
2. The undersigned agrees to indemnify and hold harmless the IEEE from any damage or expense that may arise in the event of a breach of any of the warranties set forth above.
3. The undersigned agrees that publication with IEEE is subject to the policies and procedures of the IEEE PSPB Operations Manual.
4. In the event the above work is not accepted and published by the IEEE or is withdrawn by the author(s) before acceptance by the IEEE, the foregoing copyright transfer shall be null and void. In this case, IEEE will retain a copy of the manuscript for internal administrative/record-keeping purposes.
5. For jointly authored Works, all joint authors should sign, or one of the authors should sign as authorized agent for the others.
6. The author hereby warrants that the Work and Presentation (collectively, the "Materials") are original and that he/she is the author of the Materials. To the extent the Materials incorporate text passages, figures, data or other material from the works of others, the author has obtained any necessary permissions. Where necessary, the author has obtained all third party permissions and consents to grant the license above and has provided copies of such permissions and consents to IEEE

You have indicated that you DO wish to have video/audio recordings made of your conference presentation under terms and conditions set forth in "Consent and Release."

CONSENT AND RELEASE

1. In the event the author makes a presentation based upon the Work at a conference hosted or sponsored in whole or in part by the IEEE, the author, in consideration for his/her participation in the conference, hereby grants the IEEE the unlimited, worldwide, irrevocable permission to use, distribute, publish, license, exhibit, record, digitize, broadcast, reproduce and archive, in any format or medium, whether now known or hereafter developed: (a) his/her presentation and comments at the conference; (b) any written materials or multimedia files used in connection with his/her presentation; and (c) any recorded interviews of him/her (collectively, the "Presentation"). The permission granted includes the transcription and reproduction of the Presentation for inclusion in products sold or distributed by IEEE and live or recorded broadcast of the Presentation during or after the conference.
2. In connection with the permission granted in Section 1, the author hereby grants IEEE the unlimited, worldwide, irrevocable right to use his/her name, picture, likeness, voice and biographical information as part of the advertisement, distribution and sale of products incorporating the Work or Presentation, and releases IEEE from any claim based on right of privacy or publicity.

BY TYPING IN YOUR FULL NAME BELOW AND CLICKING THE SUBMIT BUTTON, YOU CERTIFY THAT SUCH ACTION CONSTITUTES YOUR ELECTRONIC SIGNATURE TO THIS FORM IN ACCORDANCE WITH UNITED STATES LAW, WHICH AUTHORIZES ELECTRONIC SIGNATURE BY AUTHENTICATED REQUEST FROM A USER OVER THE INTERNET AS A VALID SUBSTITUTE FOR A WRITTEN SIGNATURE.

Basem Assiri	03-06-2016
Signature	**Date (dd-mm-yyyy)**

Information for Authors

AUTHOR RESPONSIBILITIES

The IEEE distributes its technical publications throughout the world and wants to ensure that the material submitted to its publications is properly available to the readership of those publications. Authors must ensure that their Work meets the requirements as stated in section 8.2.1 of the IEEE PSPB Operations Manual, including provisions covering originality, authorship, author responsibilities and author misconduct. More information on IEEE's publishing policies may be found at http://www.ieee.org/publications_standards/publications/rights/authorrightsresponsibilities.html Authors are advised especially of IEEE PSPB Operations Manual section 8.2.1.B12: "It is the responsibility of the authors, not the IEEE, to determine whether disclosure of their material requires the prior consent of other parties and, if so, to obtain it." Authors are also advised of IEEE PSPB Operations Manual section 8.1.1B: "Statements and opinions given in work published by the IEEE are the expression of the authors."

RETAINED RIGHTS/TERMS AND CONDITIONS

- Authors/employers retain all proprietary rights in any process, procedure, or article of manufacture described in the Work.
- Authors/employers may reproduce or authorize others to reproduce the Work, material extracted verbatim from the Work, or derivative works for the author's personal use or for company use, provided that the source and the IEEE copyright notice are indicated, the copies are not used in any way that implies IEEE endorsement of a product or service of any employer, and the copies themselves are not offered for sale.
- Although authors are permitted to re-use all or portions of the Work in other works, this does not include granting third-party requests for reprinting, republishing, or other types of re-use.The IEEE Intellectual Property Rights office must handle all such third-party requests.
- Authors whose work was performed under a grant from a government funding agency are free to fulfill any deposit mandates from that funding agency.

AUTHOR ONLINE USE

- **Personal Servers**. Authors and/or their employers shall have the right to post the accepted version of IEEE-copyrighted articles on their own personal servers or the servers of their institutions or employers without permission from IEEE, provided that the posted version includes a prominently displayed IEEE copyright notice and, when published, a full citation to the original IEEE publication, including a link to the article abstract in IEEE Xplore. Authors shall not post the final, published versions of their papers.
- **Classroom or Internal Training Use.** An author is expressly permitted to post any portion of the accepted version of his/her own IEEE-copyrighted articles on the author's personal web site or the servers of the author's institution or company in connection with the author's teaching, training, or work responsibilities, provided that the appropriate copyright, credit, and reuse notices appear prominently with the posted material. Examples of permitted uses are lecture materials, course packs, e-reserves, conference presentations, or in-house training courses.
- **Electronic Preprints.** Before submitting an article to an IEEE publication, authors frequently post their manuscripts to their own web site, their employer's site, or to another server that invites constructive comment from colleagues. Upon submission of an article to IEEE, an author is required to transfer copyright in the article to IEEE, and the author must update any previously posted version of the article with a prominently displayed IEEE copyright notice. Upon publication of an article by the IEEE, the author must replace any previously posted electronic versions of the article with either (1) the full citation to the

Vita

Basem Ibrahim Assiri was born in Khamis Mushayt City, Saudi Arabia. In 2007, he finished his undergraduate studies in Computer Science at King Khalid University in Saudi Arabia. In 2009 he earned a master's degree in Computer Science (Major: Software Engineering) from the University of Wollongong in Australia. He worked as a Lecturer at Jazan University in Saudi Arabia for two years. In August 2012, he came to Louisiana State University to pursue graduate studies in Computer Science (Area of Concentration: Parallel and Distributed Computing). He is currently a candidate for the degree of Doctor of Philosophy in Computer Science, which will be awarded in August 2016. In fact, Jazan University sponsors Basem Assiri in his PhD journey. When he obtains his degree he has to work in there for the same amount of time he spent on the PhD journey. In addition, parts of this research have been converted into paper form for publishing:

- Transactional Memory Scheduling Using Machine Learning Techniques (Chapter 4), has been published in the 24^{th} Euromicro International Conference in Parallel, Distributed and Network-Based Processing (PDP 2016).
- Approximately Opaque Multi-version Permissive Transactional Memory (Chapter 5 and 8) has been published in the following:
    i. 12^{th} International Workshop on Scheduling and Resource Management for Parallel and Distributed Systems (SRMPDS 2016) - in conjunction with International Conference on Parallel Processing (ICPP 1016).
    ii. 30^{th} IEEE International Parallel and Distributed Processing Symposium - PhD Forum (IPDPS 2016 PhD Forum).