Relaxing Concurrency Control
in Transactional Memory

by

Utku Aydonat

A thesis submitted in conformity with the requirements
for the degree of Doctor of Philosophy
Graduate Department of The Edward S. Rogers Department of Electrical and
Computer Engineering
University of Toronto

Copyright © 2011 by Utku Aydonat
Abstract

Relaxing Concurrency Control
in Transactional Memory

Utku Aydonat
Doctor of Philosophy

Graduate Department of The Edward S. Rogers Department of Electrical and Computer Engineering
University of Toronto

2011

Transactional memory (TM) systems have gained considerable popularity in the last decade driven by the increased demand for tools that ease parallel programming. TM eliminates the need for user-locks that protect accesses to shared data. It offers performance close to that of fine-grain locking with the programming simplicity of coarse-grain locking. Today’s TM systems implement the two-phase-locking (2PL) algorithm which aborts transactions every time a conflict occurs. 2PL is a simple algorithm that provides fast transactional operations. However, it limits concurrency in applications with high contention because it increases the rate of aborts. We propose the use of a more relaxed concurrency control algorithm to provide better concurrency. This algorithm is based on the conflict-serializability (CS) model. Unlike 2PL, it allows some transactions to commit successfully even when they make conflicting accesses. We implement this algorithm both in a software TM system as well as in a simulator of a hardware TM system. Our evaluation using TM benchmarks shows that the algorithm improves the performance of applications with long transactions and high abort rates. Performance is improved by up to 299% in the software TM, and up to 66% in the hardware simulator. We argue that these improvements come with little additional complexity and require no changes to the transactional programming model. This makes our implementation feasible.
## Contents

1 Introduction ................................................................. 1
   1.1 Thesis Contributions ............................................. 6
   1.2 Thesis Organization .............................................. 8

2 Background ..................................................................... 9
   2.1 Transactional Memory .............................................. 9
   2.2 STM Design Space .................................................. 11
      2.2.1 Liveness ....................................................... 11
      2.2.2 Granularity .................................................. 12
      2.2.3 Write Target ................................................ 13
      2.2.4 Acquiring for Writes ......................................... 13
      2.2.5 Visibility of Reads .......................................... 14
      2.2.6 Contention Management (CM) ............................. 14
   2.3 HTM Design Space .................................................. 15
      2.3.1 Conflict Detection (CD) ...................................... 15
      2.3.2 Version Management (VM) ................................. 16
      2.3.3 Conflict Management (CM) ............................... 16
      2.3.4 Classification of TM Systems .............................. 17

3 Concurrency Control in Transactional Memory .................. 18
   3.1 The Schedule ........................................................ 19
3.2 The Conflict ................................................................. 19
3.3 Consistency and Concurrency Control ............................... 20
3.4 Concurrency Control in Transactional Memory .................... 23
3.5 Conflict Serializability ................................................... 26
3.6 Implications for TM Systems ........................................... 28

4 Serializability Order Numbers ............................................ 30
4.1 Overview ................................................................. 30
4.2 Formulation ............................................................. 32
4.3 Read-Numbers and Write-Numbers ................................... 34
4.4 Safety Analysis .......................................................... 36
4.5 Classification and limitations ........................................... 37
4.6 Implementation Requirements ......................................... 39
4.7 Examples ................................................................. 40

5 Relaxed Concurrency in Software ....................................... 44
5.1 TSTM Overview ........................................................ 44
5.2 Programming Model ..................................................... 45
  5.2.1 Word-based Programming Model ................................. 46
  5.2.2 Object-based Programming Model ............................... 46
5.3 Data Structures ........................................................ 47
  5.3.1 Descriptor Class ...................................................... 47
  5.3.2 SharedItem Class .................................................... 47
  5.3.3 Global Readers Table (GRT) ...................................... 48
5.4 Transactional Actions ................................................... 50
  5.4.1 sharedRead .......................................................... 50
  5.4.2 sharedWrite .......................................................... 51
  5.4.3 commit ............................................................... 51
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.5</td>
<td>Race Conditions</td>
<td>54</td>
</tr>
<tr>
<td>5.5.1</td>
<td>Committing Writer vs. Committing Writer</td>
<td>54</td>
</tr>
<tr>
<td>5.5.2</td>
<td>Active Reader vs. Committing Writer</td>
<td>54</td>
</tr>
<tr>
<td>5.5.3</td>
<td>Committing Reader vs. Committing Writer</td>
<td>56</td>
</tr>
<tr>
<td>5.6</td>
<td>Performance</td>
<td>56</td>
</tr>
<tr>
<td>5.7</td>
<td>Optimizations</td>
<td>57</td>
</tr>
<tr>
<td>5.7.1</td>
<td>Retiring Transactions</td>
<td>57</td>
</tr>
<tr>
<td>5.7.2</td>
<td>The SON Table and The Aborts List</td>
<td>58</td>
</tr>
<tr>
<td>5.7.3</td>
<td>Active Readers in the GRT</td>
<td>60</td>
</tr>
<tr>
<td>5.7.4</td>
<td>Memory Operations and Atomicity</td>
<td>60</td>
</tr>
<tr>
<td>5.8</td>
<td>Adaptive TSTM</td>
<td>61</td>
</tr>
<tr>
<td>6</td>
<td>Relaxed Concurrency in Hardware</td>
<td>63</td>
</tr>
<tr>
<td>6.1</td>
<td>Base Hardware TM Support</td>
<td>63</td>
</tr>
<tr>
<td>6.2</td>
<td>Hardware Components</td>
<td>65</td>
</tr>
<tr>
<td>6.2.1</td>
<td>SON Registers</td>
<td>66</td>
</tr>
<tr>
<td>6.2.2</td>
<td>The Virtual Write-Number Table</td>
<td>66</td>
</tr>
<tr>
<td>6.2.3</td>
<td>The Read History Table</td>
<td>68</td>
</tr>
<tr>
<td>6.2.4</td>
<td>Conflict Flags</td>
<td>70</td>
</tr>
<tr>
<td>6.3</td>
<td>Transactional Actions</td>
<td>71</td>
</tr>
<tr>
<td>6.4</td>
<td>Race Conditions</td>
<td>75</td>
</tr>
<tr>
<td>6.5</td>
<td>Optimizations</td>
<td>75</td>
</tr>
<tr>
<td>6.6</td>
<td>Qualitative Performance Analysis</td>
<td>76</td>
</tr>
<tr>
<td>6.7</td>
<td>Implementation in Directory Based Systems</td>
<td>77</td>
</tr>
<tr>
<td>6.7.1</td>
<td>Issues</td>
<td>77</td>
</tr>
<tr>
<td>6.7.2</td>
<td>Hardware Support</td>
<td>78</td>
</tr>
</tbody>
</table>
7 Experimental Evaluation

7.1 Evaluation of TSTM

7.1.1 Evaluated STM Systems

7.1.2 TSTM Parameters

7.1.3 STM Experimental Platform

7.1.4 STM Evaluation Metrics

7.1.5 STM Benchmarks

7.1.6 Experimental Results

Comparative Performance of the STM Systems

Comparative Performance of the Benchmarks

Breakdown of Execution

7.2 Evaluation of SONTM

7.2.1 Evaluated HTM Systems

7.2.2 HTM Experimental Platform

7.2.3 HTM Evaluation Metrics

7.2.4 HTM Benchmarks

7.2.5 Experimental Results

The impact of the SON Support

The breakdown of execution

The impact of SONTM parameters

Summary of the Results

8 Related Work

8.1 Transaction Management in Databases

8.2 Conflict-Serializability in Transactional Memory

8.3 Other Approaches in TM Systems

9 Conclusions
List of Tables

6.1 Flags that are set for each type of conflict. ........................................ 71
6.2 Actions taken by a processor when it receives a SON broadcast message. .... 74
6.3 Flags that are set for each type of conflict. ........................................ 80
6.4 How $uptLower$ and $uptUpper$ bit vectors are set at commit time. ............. 81
6.5 Actions taken by processor $i$ when it receives a SON broadcast message from processor $k$. .......................................................... 81

7.1 The implementation parameters of TSTM. ........................................ 85
7.2 Specifications of the experimental platform for evaluating STM systems. .... 87
7.3 Benchmark input parameters for STM systems. .................................... 90
7.4 Benchmark characteristics measured with the base STM system. .............. 90
7.5 Experimental results on 16 threads. .................................................. 99
7.6 Hardware model parameters used for evaluating HTM systems. ............... 104
7.7 Benchmark input parameters for HTM systems. .................................... 106
7.8 Benchmark characteristics measured with the $2pl-EL$ HTM system. ......... 107
7.9 Normalized execution breakdown on 8 processors. ............................... 113
List of Figures

3.1 The schedule of three transactions accessing data items A and B. 19
3.2 Serializable and non-serializable schedules. 22
3.3 2PL implementation of TM systems. 24
3.4 2PL concurrency algorithm causes aborts and/or delays. 27
4.1 All nine possible schedules for conflicting read and write actions under lazy VM. 35
4.2 Example schedules demonstrating the SON algorithm. 41
5.1 The Global Readers Table. 49
5.2 TSTM data structures and transactional actions for sequential execution. 52
5.3 Active reader transaction vs. committing writer race condition. 55
5.4 Concurrent notifyReaders. 59
5.5 The algorithm for adaptive concurrency control. 62
6.1 Hardware support overview. 67
6.2 History table. 69
6.3 SONTM transactional actions. 72
6.4 Conflicts missed by the directory coherence protocol in an EL system. 78
7.1 Abort rate and Xact Time distribution of the benchmarks measured with the base STM system. 91
7.2 Speedup and abort rates for list, vacation, and labyrinth. 96
7.3 Speedup and abort rates for labyrinth-nop, yada, and intruder. 97
7.4 Speedup and abort rates for genome, kmeans, and ssca2. . . . . . . . . . . . 98
7.5 The breakdown of execution for base and cs systems. . . . . . . . . . . . . 101
7.6 Abort rate and transaction cycles distribution of the benchmarks measured with
the 2p1-EL HTM system. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 108
7.7 Normalized total execution cycles and abort rates on 8 processors. . . . . . . 111
7.8 The impact of selected address bits and the size of the virtual table. . . . . . . 115
## Table of Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Term</th>
</tr>
</thead>
<tbody>
<tr>
<td>2PL</td>
<td>Two-Phase Locking</td>
</tr>
<tr>
<td>CD</td>
<td>Conflict Detection</td>
</tr>
<tr>
<td>CM</td>
<td>Contention Management</td>
</tr>
<tr>
<td>CS</td>
<td>Conflict-Serializability</td>
</tr>
<tr>
<td>DBMS</td>
<td>Database Management System</td>
</tr>
<tr>
<td>EE</td>
<td>Eager Conflict Detection, Eager Version Management</td>
</tr>
<tr>
<td>EL</td>
<td>Eager Conflict Detection, Lazy Version Management</td>
</tr>
<tr>
<td>GRT</td>
<td>Global Readers Table</td>
</tr>
<tr>
<td>HTM</td>
<td>Hardware Transactional Memory</td>
</tr>
<tr>
<td>LL</td>
<td>Lazy Conflict Detection, Lazy Version Management</td>
</tr>
<tr>
<td>SON</td>
<td>Serializability Order Number</td>
</tr>
<tr>
<td>STM</td>
<td>Software Transactional Memory</td>
</tr>
<tr>
<td>TM</td>
<td>Transactional Memory</td>
</tr>
<tr>
<td>VM</td>
<td>Version Management</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

One of the biggest challenges in writing multi-threaded programs is to properly synchronize accesses to shared data in order to preserve correctness. This requires programmers to carefully analyze data accesses of each parallel thread and to use the appropriate synchronization primitives at critical sections where shared data is accessed. The most popular of such synchronization primitives are locks that allow only one thread to execute a particular code region at a time. However, this classic lock-based synchronization approach suffers from several drawbacks [30]. First, programmers must associate each datum that needs to be protected with a lock and obey this association throughout the entire program. Second, programmers need to carefully choose the granularity at which the locks are used. If locks are used at a coarse granularity, execution may be serialized even if there is no actual data sharing, reducing concurrency. Using locks at a fine granularity may expose more concurrency, but it may also incur high overheads due to locking and unlocking operations. More importantly, the fine-grain use of locks is difficult to program and may easily cause deadlocks. Choosing the right balance between coarse-grain and fine-grain use of locks is time-consuming and error-prone. Traditional locking mechanisms may also cause other problems such as priority inversion, convoying and lack of fault tolerance [30].
Transaction memory (TM) has been proposed as a solution to such synchronization problems in multi-threaded programs [19]. In this model, user-locks are completely eliminated. A programmer declares each critical section as a transaction and re-directs data accesses within transactions to the TM system. The underlying TM system keeps track of shared data accesses of all transactions to determine which accesses conflict with one another. The system makes sure that transactions appear to run in isolation and atomically, preserving the consistency of shared data. If the consistency condition is met, a transaction commits by updating the system state permanently. If the consistency of shared data cannot be guaranteed, the transaction aborts by discarding or undoing any updates it made to the system state.

The main promise of TM is to facilitate parallel programming by eliminating the need for user-locks while still providing performance that is close to that of carefully designed fine-grain locking [30]. With the high demand for tools that ease developing multi-threaded programs that run efficiently on multi-core systems, TM systems gained considerable popularity in recent years. Several TM systems, either implemented in software [3, 12, 14, 18, 28, 31, 39, 44, 45] or in hardware [7, 15, 19, 22, 34, 38, 48], have been proposed. The design space of TM has been extensively investigated. Consequently, the research community has gained a better understanding of the design tradeoffs that exist. This not only resulted in significant performance improvements, but it also triggered research in TM language support [1, 6, 8, 42] and TM compiler support [17, 44]. Although the community is still experimenting with different hardware and software approaches, TM research has come long way since the first hardware [19] and software [45] implementations that were introduced more than a decade ago. The benefits of TM are widely believed-in and many researchers expect that some type of TM support will exist in future languages, compilers and hardware systems [1, 6, 8, 17, 42, 44]. However, despite all these promising developments, TM systems still suffer from some drawbacks such as bounded transactions in hardware [2], the lack of clear semantics for nested transactions [35], handling irreversible operations inside transactions, such as I/O, system calls, and memory allocation and de-allocation [16], and overcoming high overheads in
One of the major problems for TM systems has been low performance for applications that contain long-running (or simply long) transactions and high degrees of data sharing, i.e. high contention. In this thesis, we argue that this is due to the strict algorithms implemented by TM systems in order to preserve the consistency of shared data. TM systems ensure this consistency by implementing the serializability consistency model. That is, TM systems ensure that after transactions are executed concurrently, the state of the shared data and the outcome of transactions is the same as if these transactions were executed serially in some order.

TM systems ensure serializability by implementing algorithms that determine what actions must be taken when a conflict occurs. These algorithms are called concurrency control algorithms. Existing TM systems (both hardware and software) implement the Two-Phase-Locking (2PL) algorithm [5]. In this algorithm, transactions are simply not allowed to perform conflicting actions. If a transaction attempts a conflicting access, the conflict is handled by aborting one of the conflicting transactions, or by delaying the access until one of the conflicting transactions commits. 2PL is a simple algorithm with straightforward implementations, and hence, it provides fast transactional actions. However, 2PL limits concurrency because every single conflict causes an abort or a delay. This becomes a significant performance bottleneck for applications that contain long transactions with high contention [3, 4, 5]. These applications suffer from frequent aborts or delays, and the performance penalty of aborting and retrying long transactions is high. In order to alleviate these challenges, the use of manual techniques such as early-release [18, 49] and open nesting [6, 35] have been proposed. Regrettably, these manual techniques simply aim to eliminate the conflicts that cause unnecessary aborts or delays and, hence, they require considerable programming effort to obtain good performance and to ensure correctness.

We argue that 2PL can be an overly conservative algorithm for guaranteeing serializability in TM systems. We propose to use a more relaxed algorithm for enforcing serializability, which is based on a special type of serializability called conflict-serializability (CS) [9]. Under CS,
some transactions would be allowed to successfully commit even if they make conflicting accesses during their concurrent execution. This is possible if we can find a serial execution order of transactions such that the order of the conflicting accesses of the transactions is the same in this serial execution and in the concurrent execution. This guarantees that, during the concurrent execution, transactions read and write the same values of shared data as they do during the serial execution, hence, these values as well as the final state of the system are consistent. In the remainder of this thesis, we generally refer to the serializability model implemented by the relaxed algorithms that allow conflicting transactions to commit successfully as the CS model. This contrasts to the traditional 2PL algorithm that does not allow conflicting transactions.

The CS model has the potential to reduce the number of aborts and delayed transactional accesses, and provide better concurrency than 2PL. This can increase the performance particularly for applications that contain long transactions with high contention. However, CS is more complex than 2PL. A naive implementation can incur high overheads and slow down transactional actions.

We propose to use an efficient algorithm for implementing the CS model in software and hardware TM systems. This algorithm serially orders transactions as they execute by assigning them Serializability Order Numbers (SONs). The SON of a transaction represents the order of that transaction in a serial ordering of all the past and future transactions. The TM system keeps track of the transactional data accesses and assigns SONs to transactions when they commit based on the order of their conflicting accesses. We refer to this algorithm as the SON algorithm. Similar to CS model that it implements, the SON algorithm also allows transactions to complete even when conflicting accesses to shared data are present. Therefore, transactions can commit when they would have stalled or aborted with the use of 2PL, which can lead to more concurrency and to better performance.

However, although the SON algorithm implements CS efficiently, it still incurs some overheads. The overheads of the SON algorithm may slow down applications that already have low abort rates with 2PL. Therefore, we also introduce an adaptive approach that uses 2PL when
the abort rate is low and switches to using CS when the rate is high. The adaptive approach allows selecting the best concurrency algorithm depending on the characteristics of the target application, and thus producing performance that matches the best performance of 2PL and CS.

In order to evaluate the use of CS in TM systems and its impact on performance, we first implement our algorithm in an obstruction-free software TM system. Our evaluation using standard TM benchmarks shows that the use of CS does indeed result in significantly lower abort rates compared to the use of 2PL. This leads to better performance for applications that have long transactions and high contention. For instance, the use of CS improves the throughput of transactions in a linked-list benchmark by 114% with 16 threads compared to 2PL. The throughput of another benchmark (vacation) is improved by 199%. Furthermore, the performances of these benchmarks scale well with CS, but not with 2PL. We also compare the performance of our adaptive system to that of a state-of-the-art blocking STM system (TL2) and show that the throughput of transactions on our system exceeds that of TL2 for most applications.

In order to show that hardware systems can also benefit from lowering the abort rates using CS, we design the support necessary to implement the SON algorithm in hardware. This novel hardware TM system requires little additional hardware complexity on top of existing fundamental TM components used for conflict detection and isolation. The additional hardware consists of a few simple components per processor: a small read history table, a set of registers to hold SONs and related data, and a set of flags for keeping track of conflicts. In addition, a table is maintained in virtual memory to keep track of the SONs of transactions for each shared address. The use of these components introduces some overheads that reflect cache misses due to the accesses to the virtual table and messages used for communicating SONs, commit events and conflicts among transactions.

We implement the SON algorithm in a simulator of a hardware TM system. Our evaluation using this simulator shows that our HTM system outperforms the base HTM system that uses
2PL. Performance improves by 29% on average over nine benchmarks. In fact, the throughputs of benchmarks with long transactions and high abort rates improves by 32% to 66%. Thus, our simple hardware extensions are able to reap the benefit of CS. Further, performance is not significantly worse than an ideal CS implementation that incurs no overheads. The difference in performance is below 6% on average over nine benchmarks, which leads us to believe that our implementation is efficient. Finally, the performance of our system is not significantly sensitive to the parameters of the proposed hardware components (e.g., size of tables, bits used for hashing, etc.). Indeed, our evaluation shows that it is possible to select these parameters to work well for a wide range of benchmarks. This leads us to believe that our implementation is feasible.

We also evaluate the performance of another TM system (DATM) that implements CS and show that its performance is comparable to that of our system. However, our implementation is superior because it only requires simple additional hardware support as opposed to complex changes to caches and their coherence protocols required by the other approaches.

Finally, both our software and hardware prototype systems do not support nesting of transactions and allow only a single transaction to be executed by each thread at a time. This allowed us to implement the SON algorithm most efficiently, and to focus on the implementation and the evaluation of the use of CS.

### 1.1 Thesis Contributions

This thesis improves the state of TM research by recognizing that the traditional concurrency algorithm implemented by the TM systems (i.e. 2PL) is overly conservative particularly for applications with long transactions and high contention. Thus, the thesis describes an algorithm that can efficiently be implemented both in software and hardware TM systems in order to reduce the abort rates of transactions and improve the performance in such applications. More specifically, this thesis makes the following contributions.
1. *Identifying the strictness of 2PL in TM systems.* Our work is the first to recognize that the use of a more relaxed algorithm of serializability can improve the performance of applications with long transactions and high abort rates. We show that the 2PL concurrency algorithm results in higher than necessary abort rates for such applications.

2. *An algorithm for efficiently implementing CS in TM systems.* We introduce the SON algorithm which extends the optimistic concurrency control algorithms proposed for database management systems. We design the algorithm to efficiently implement the CS model in software and hardware TM systems. We define the transactional operations, and show that the algorithm is safe and more relaxed than 2PL.

3. *An implementation of CS in a software TM system.* We introduce the first software TM system in the literature that implements the CS model using the SON algorithm. We present the necessary algorithms for transactional actions and the required data structures.

4. *An adaptive TM implementation that leverages the advantages of both 2PL and CS.* We present the first adaptive algorithm in the literature that can be used to choose the best concurrency algorithm depending on the target application.

5. *The design of a hardware TM system that implements CS.* We introduce a novel hardware TM system that implements CS fully in hardware with minimal hardware support and no changes to the existing cache coherence protocols.

6. *An experimental evaluation of both the software and hardware implementations.* We evaluate using a multi-processor SMP system and a commercial full system simulator the effectiveness of the SON algorithm in reducing the abort rates and in improving the performance of software and hardware systems.
1.2 Thesis Organization

The remainder of this thesis is organized as follows. Chapter 2 gives a brief a background on transactional memory and related concepts. Chapter 3 describes the concurrency control algorithms implemented by TM systems. Chapter 4 presents our SON algorithm for efficiently implementing CS in TM systems. Chapter 5 introduces our software TM system that implements the SON algorithm. Chapter 6 introduces our hardware TM system that implements the SON algorithm. Chapter 7 presents and discusses the experimental results for the software and hardware TM systems. Chapter 8 discusses the related work in literature. Chapter 9 concludes the thesis.
Chapter 2

Background

This chapter introduces the transactional memory paradigm. Section 2.1 defines transactional memory concepts and describes its working principles. Section 2.2 presents the design space for software transactional memory implementations. Section 2.3 describes the design space for hardware transactional memory implementations.

2.1 Transactional Memory

A transaction is a sequence of instructions that access and/or modify shared data and has the atomicity, consistency and isolation properties [19]. Respectively, these properties imply that (1) the effects of transactions appear to be performed instantaneously all together, or none of them are performed, (2) the state of shared data is always consistent, and (3) data modified by a transaction cannot be seen by other transactions until the transaction successfully completes. A TM system guarantees that these properties hold for all transactions at run-time. If a transaction completes without violating the consistency property, it commits by permanently changing the state of the shared data and making these changes visible to other transactions. If a transaction violates the consistency property, it aborts by undoing or discarding changes it made to the system state. The TM system may choose to restart aborted transactions immediately or apply an exponential backoff mechanism [20].
A transaction can perform a number of reads and writes on a shared data item (e.g. a shared address, a shared object, etc.). These accesses are referred to as *transactional actions*.

A *conflict* is said to exist between two or more actions if (1) the actions belong to different transactions, (2) the actions access the same data item, (3) at least one of the actions is a write. Conflicts are a potential source for inconsistency among transactions. Most TM systems choose to abort at least one the conflicting transactions or delay the transactional actions of conflicting transactions to preserve consistency. Chapter 3 discusses the approaches that can be used to ensure the consistency of shared data.

Transactions provide several advantages over direct use of locking mechanisms. First, programmers no longer need to worry about associating locks with shared data; TM ensures that code blocks marked as transactions appear to execute atomically. This makes it easier to write parallel programs. Second, choosing the right granularity of locks is no longer an issue because the TM system keeps track of accesses to each individual datum, thus giving the benefit of fine-grain locking. Third, deadlocks do not occur in TM systems because transactions can be aborted once mutual dependence is detected, discarding the modifications of the aborted transactions. Fourth, convoying and priority inversion can be avoided by allowing high-priority transactions to abort low-priority transactions and preventing low-priority transactions from aborting high-priority ones [30]. Fifth, fault tolerance comes naturally as the modifications made by transactions are reversible, providing consistent state in case of system crashes.

Any TM system must implement the following mechanisms: (1) isolation of stores, (2) conflict detection, (3) atomic commits, and (4) rollback of stores. Isolation of stores is required to ensure the isolation property, i.e., speculative modifications made by each transaction are not visible to other transactions until commit time. Conflict detection is the mechanism of inspecting transactional actions at run-time in order to detect conflicts. Atomic commit is achieved when all speculative modifications of a transaction become visible to other transactions instantly. Rollback of stores is needed to discard all the modifications of a transaction when it aborts so that the state of the shared data remains as if the transaction has never executed.
The above mechanisms of TM systems can be implemented in software, in hardware, or partly in hardware and software. These implementations are respectively referred to as Software Transactional Memory (STM), Hardware Transactional Memory (HTM), and Hybrid Transactional Memory systems. STM systems [3, 12, 14, 18, 28, 31, 39, 44, 45] are typically implemented as libraries that provide functionality for starting and committing transactions. Transactional actions are kept track of using special function calls inserted into the code by the programmer before each access to shared data. HTM systems [7, 15, 19, 22, 34, 38, 48] require additional instructions in the ISA to support starting and committing transactions. Conflict detection is typically achieved by taking advantage of the existing cache coherence protocols. Hybrid TM systems [2, 4, 10] are implemented as a combination of software libraries and hardware components. These systems usually keep transaction metadata in software and use hardware mechanisms to keep track of conflicts.

The mechanisms of TM systems may be implemented in a number of ways and the various implementations give rise to a large and complicated design space. The remainder of this chapter presents the design space for STM and HTM systems.

### 2.2 STM Design Space

#### 2.2.1 Liveness

STM systems can be considered as synchronization algorithms in that they manage shared data accesses of parallel processes. In general, a synchronization algorithm can be blocking or non-blocking. A blocking algorithm allows only one process (or thread) to access a concurrent object (or shared data) at a time and stops all the remaining processes. In contrast, in a non-blocking algorithm, processes do not need to wait to access a concurrent object. Non-blocking algorithms can be classified into three classes: wait-free, lock-free and obstruction-free. For some processes that contend for a number of concurrent objects: (1) a wait-free algorithm guarantees that all the processes make progress in a finite number of steps, (2) a lock-free algo-
Algorithm guarantees that at least one process makes progress, and (3) an obstruction-free algorithm guarantees that a single process makes progress in the absence of contention. Among the proposed STM systems, DSTM [18] and RSTM [31] are obstruction-free because they allow a conflicting transaction to abort another. McRT-STM [44] and TL2 [12] are blocking because they require transactions to acquire the locks associated with the data they access. In contrast, FSTM [14] is lock-free because it allows a conflicting transaction to help another to commit its results.

In terms of the limitations a synchronization algorithm imposes on progress, wait-free algorithms are the most relaxed, whereas obstruction-free algorithms are the most strict. In terms of the complexity of their implementations, blocking algorithms are generally the simplest, obstruction-free algorithms are moderately simple; and lock-free algorithms are the most complex ones. In exchange for the simplicity, blocking STM algorithms (e.g. McRT-STM [44] and TL2 [12]) allow deadlocks and obstruction-free algorithms (e.g. DSTM [18] and RSTM [31]) allow live-locks. On the other hand, neither deadlocks nor live-locks are possible with lock-free algorithms (e.g. FSTM [14]).

### 2.2.2 Granularity

STM systems typically detect conflicts at two different granularities: the object-level and the word-level. With object-level conflict detection, the STM system guarantees that two transactions do not access the same \textit{C++/Java class object} in a conflicting way. In contrast, with the word-level conflict detection, the STM system gives the same guarantee for two transactions accessing the same \textit{memory word}. Object-level detection avoids the overhead of bookkeeping for every word, but may cause false conflicts. Further, object-level conflict detection is easier to implement in managed environments (such as Java) because of the metadata already associated with each object at run-time. Among the existing STM systems, McRT-STM [44] and TL2 [12] use word-level conflict detection, DSTM [18] use object-level conflict detection, and RSTM [31] can be configured to use either word-level or object-level conflict detection.
2.2.3 Write Target

A TM system can implement one of two mechanisms to store the modifications of transactions: write-buffering and undo-logging. In write-buffering, transactions buffer their modifications in a temporary storage and apply them to the memory permanently when they commit. In undo-logging, a transaction directly updates the memory and keeps a log to rollback the memory state if the transaction aborts. Write-buffering makes aborts faster while undo-logging makes commits faster. Thus, it is advantageous to use undo-logging when transactions are likely to commit and to use write-buffering when transactions are likely to abort. Undo-logging also makes transactional reads faster because data is directly read from the memory. In contrast, with write-buffering, data is first searched for in the transactional buffer. Further, undo-logging only allows one of the conflicting transactions to proceed so that transactional writes will not interfere with each other. Among existing STM systems, DSTM [18] implements write-buffering, McRT-STM [44] implements undo-logging and TL2 [12] and RSTM [31] can be configured to use either one of the mechanisms.

2.2.4 Acquiring for Writes

STM systems guarantee atomic commits for transactional writes by acquiring a lock associated with each shared object (or word) they write to. STM systems may implement two types of policies on how these locks are acquired: eager acquire and lazy acquire. In eager acquire, a transaction gets the ownership of the object during execution just before writing to the object, announcing to the other transactions that if they try access the object a conflict will arise. In lazy acquire, transactions write to concurrent objects independently, without claiming any kind of ownership. Conflicts are detected at commit time when transactions temporarily acquire the ownerships of the objects to commit their updates to the memory. Eager acquire allows early detection of conflicts: a conflict will be detected immediately if a transaction tries to access an object which was already acquired by another transaction. However, eager acquire
may cause unnecessary aborts because a transaction may abort other transactions before it is aborted itself. Generally, eager acquire performs better by preventing useless work, whereas lazy acquire tends to perform better when there is a large number of threads, increasing the chances of unnecessary aborts. Among the existing STM systems, DSTM [18] and McRT-STM [44] use eager acquire, and RSTM [31] and TL2 [12] can be configured to use either one of the mechanisms.

2.2.5 Visibility of Reads

Similar to acquiring for writes, an STM system can implement two types of policies for detecting conflicts on transactional reads: visible reads and invisible reads. With visible reads, transactions are aware of any object (or memory word) read by other transactions. Conflicts are immediately detected if a transaction tries to write to an object that was already read by another transaction. In contrast, with invisible reads, objects are read transparently to other transactions. To detect conflicts, transactions periodically validate all their reads by verifying that the data they read is still the most up to date data in memory. The visible reads technique allows quick detection of conflicts. However, it increases cache misses because transactions update a shared data structure when they read shared data [30]. In contrast, the invisible reads technique requires costly read validation, as mentioned above. Among the existing STM systems, DSTM [18], McRT-STM [44] and TL2 [12] use invisible reads while RSTM [31] can be configured to use either one of the techniques.

2.2.6 Contention Management (CM)

STM systems implement algorithms that determine what actions should be taken when a conflict is detected. These mechanisms are called contention management systems. Policies for contention management vary from aggressive to polite. In an aggressive policy, a transaction that performs a shared access immediately aborts another transaction if it detects a conflict. In
contrast, in a polite policy, transactions stall their shared accesses and wait before they abort the other conflicting transaction, possibly allowing it to commit successfully in the meantime. This is possible because conflicts with already committed transactions do not cause inconsistency of shared data in the existing TM systems.

2.3 HTM Design Space

HTM systems are typically obstruction-free because transactions are aborted when a conflict is detected. Further, they typically use cache line granularity for conflict detection because they take advantage of the cache coherence protocols. Below, we discuss some other dimensions of the HTM design space.

2.3.1 Conflict Detection (CD)

Similar to how written objects are acquired and how transactional reads are made visible or invisible in STM systems, two types of policies can be implemented in an HTM system for conflict detection: eager CD and lazy CD. With eager CD, conflicts are immediately detected with the help of the cache coherence messages as transactional accesses are issued. With lazy CD, on the other hand, conflicts are detected at commit time when transactions broadcast their updates to other transactions and/or commit them into the memory. Eager CD allows early detection of conflicts so that some transactions that will eventually abort do not waste any resources and restart as early as possible. Lazy CD delays the detection of conflicts until commit time giving time to transactions to commit successfully before any conflicts are detected. Thus, a conflict that would have been detected and taken into account under eager CD may not be detected under lazy CD. Therefore, HTM systems that implement lazy CD typically have lower abort rates.

As mentioned earlier, conflicts with already committed transactions are not taken into account by the existing TM systems.
2.3.2 Version Management (VM)

The policies for handling the writes of transactions are generally referred to as version management in the context of HTM systems. They are analogous to choosing the write target in STM systems.

There exist two version management policies: eager VM and lazy VM. With eager VM, active transactions perform their updates in place (in cache or memory); each transaction keeps an undo-log of old values. With lazy VM, old values are kept in place and updates are buffered in a private per-transaction write-buffer until commit time. At commit time, processors issue coherence requests for exclusive access to the cache lines stored in their write-buffers. These exclusive requests invalidate other copies of the cache lines, aborting any reader transactions. Once a committing transaction gains exclusive access to a cache line, it flushes the corresponding data from its transactional write-buffer to the memory. A transactional load, on the other hand, first acquires shared permissions for the cache line, then the transaction adds the block to its read-set. The actual data returned by the load is supplied from the transactional write-buffer, if present, otherwise, it is taken from memory. Lazy VM provides fast aborts because the old values are kept in place, whereas the eager VM provides fast commits because the new values are kept in place.

2.3.3 Conflict Management (CM)

When a conflict is detected, the conflict management algorithm dictates what type of action must be taken. Three types of actions can be taken in an HTM system depending on which transaction wins the conflict: committer-wins, requester-wins and requester-stalls. With the committer-wins policy, when a conflict is detected, the transaction that is committing continues its execution and all the other conflicting transactions are aborted. With the requester-wins policy, the transaction that receives a conflicting memory request for shared data aborts. With the requester-stalls policy, a memory update that conflicts with an access of another transaction
is stalled until the other conflicting transaction commits (a conflict causes an abort only if a stall may potentially cause a deadlock).

### 2.3.4 Classification of TM Systems

The design decisions described above can be combined in three valid (i.e. provides consistency) classes of HTM systems [20].

1. **Lazy CD/Lazy VM/Committer-Wins.** These systems are referred to as *LL systems*. TCC [15], Bulk [7] and FlexTM [48] are the major HTM systems of this type. In order to perform atomic writes, LL systems acquire commit tokens [32], and to ensure forward progress, they implement the committer-wins policy.

2. **Eager CD/Lazy VM/Requester-Wins.** These systems are referred to as *EL systems*. LTM [2] is an example HTM system of this type.

3. **Eager CD/Eager VM/Requester-Stalls.** These systems are referred to as *EE systems*. LogTM implementations [22, 34] are the main representatives for this type of hardware support.
Chapter 3

Concurrency Control in Transactional Memory

TM systems ensure the correctness of concurrent execution and the consistency of shared data by implementing concurrency control algorithms. In this thesis, we argue that the traditional algorithm implemented by the existing TM systems is overly conservative in that it results in loss of concurrency. This becomes a performance bottleneck and limits performance for certain types of applications. Thus, we propose the use of a more relaxed algorithm in order to provide better concurrency.

This chapter presents several important notions that are necessary for the development of our work, such as schedules and the classification of schedules. Further, it discusses how consistency is ensured for different classes of schedules. The chapter also describes the traditional 2PL concurrency control algorithm implemented by the existing TM systems and presents its weaknesses. Finally, a more relaxed concurrency control algorithm is presented and its benefits are debated.
3.1 The Schedule

The schedule is the list of the actions performed by a set of transactions in the order they happen in time. Figure 3.1 shows the schedule of three transactions, TX1, TX2, and TX3. The read and write actions of the transactions are listed from top to bottom in the order they are executed by each transaction. For instance, \( \text{Read}(\text{TX1}, A) \) represents a read action of transaction TX1 on data item A. Similarly, \( \text{Write}(\text{TX2}, B) \) represents a write action of TX2 on B. Finally, \( \text{Start}(\text{TX1}) \) represents the start time of TX1 and \( \text{Commit}(\text{TX1}) \) represents the commit action of TX1.

In the schedule, transactional operations that are not relevant for the consistency of the execution (such as non-shared data accesses, non-conflicting accesses, etc.) are represented by (...) Thus, in the above example schedule, TX1 starts, reads data item A, writes to B, and commits. Subsequently, TX2 starts, writes to A and B, and commits. Finally, TX3 starts, writes to B and then commits.

3.2 The Conflict

The definition of conflicting accesses given in Section 2.1 states that two transactional actions are conflicting if they belong to different transactions and at least one of the actions is a write. For instance, in Figure 3.1, the pair of actions \( \text{Read}(\text{TX1}, A) \) and \( \text{Write}(\text{TX2}, A) \); \( \text{Write}(\text{TX1}, B) \) and \( \text{Write}(\text{TX2}, B) \); \( \text{Write}(\text{TX1}, B) \) and \( \text{Write}(\text{TX3}, B) \); and
Write(TX2, B) and Write(TX3, B) are conflicting.

In the remainder of this thesis, writes are considered to cause conflicts only when their updates impact the consistency of shared data. That is, writes conflict with other read or write actions when their updates to memory become visible to other transactions. This definition may lead to two types of TM systems. The first is an eager VM system where the writes become visible when the write action is issued. However, such a system would break the isolation property of TM systems. The second is a lazy VM system where the writes become visible when transactions commit. The software and hardware TM implementations described in Chapter 5 and in Chapter 6 use lazy VM. This is described in detail in Section 4.6. In addition, the HTM system uses eager CD which detects conflicts when transactional loads and stores are issued. However, conflicts are still handled when transactions commit.

### 3.3 Consistency and Concurrency Control

The consistency property of a TM system ensures that any changes made to data items by a transaction remain consistent. That is, a transaction can change the data items only in ways that are possible with serial execution. In other words, every transaction transforms a consistent state (a state reachable with serial execution) to another consistent state, where a state can be defined as the values of program variables stored in permanent storage such as caches, shared memory, or the disk. Concurrent execution of transactions is a source of inconsistency and may result in incorrect execution (i.e., the application reaches an invalid state unreachable with serial execution) on a TM system.

TM systems implement consistency models to ensure that the consistency property holds for each transaction. In general, TM systems implement the serializability consistency model. Schedules can be divided into three categories in terms of serializability: serial, serializable, and non-serializable. A serial schedule is a schedule in which the actions of transactions do not interleave with each other. Serial schedules are naturally consistent and result in correct
execution. A *serializable* schedule is a schedule in which the actions of transactions do interleave with each other, however, the impact of these actions on the system state is the same as that of some serial schedule consisting of the same actions, regardless of what the initial state of the system is. That is, the outcome of the concurrent execution of transactions is *equivalent* to the outcome of a serial execution. In a serializable schedule, it can be assumed that each transaction executes instantaneously at some point, called the *serialization point*, between its start and commit. Finding a serialization point for all transactions is called a *serialization*. Finally, a *non-serializable* schedule is a schedule that is not equivalent to any serial schedule.

For instance, the schedule shown in Figure 3.1 is serial because all the actions of TX1 happen before the actions of TX2 and all the actions of TX2 happen before the actions of TX3. Further, the schedules shown in Figure 3.2 are similar to the serial schedule shown in Figure 3.1 in that they contain the same transactions and their actions, however, the order of these actions are different. In Figure 3.2(a), the actions of transactions interleave and, therefore, this schedule is not serial. However, after the three transactions are executed concurrently, the state of the system (i.e., the values of the shared items A and B) is the same as the state of the system after the serial schedule in Figure 3.1. That is, (1) TX1 reads the initial value of A before TX2 writes to it, (2) the final value of A is written by TX2, and (3) the final value of B is written by TX3. Thus, the schedule shown in Figure 3.2(a) is serializable and is equivalent to a serial schedule in which TX1 executes before TX2 and TX2 executes before TX3. This valid serialization is denoted as TX1→TX2→TX3.

In contrast, the schedule shown in Figure 3.2(b) is non-serializable. This is because, in this schedule, the final value of B is written by TX1, but not by TX3 as in the serial schedule. This means that TX1 should be serialized after TX2 and TX3 that both write to B. In any serial schedule in which TX1 executes after TX2 and TX3, TX1 should also read the value of A that is written by TX2. However, in this non-serial schedule, TX1 reads the initial value A. Thus, this schedule is not equivalent to any serial schedule that contains the same actions, and hence, it is not serializable. However, this assumes that the final value of B written by TX1 is different.
than the final value of \( B \) written by \( TX3 \). If these two transactions write the same value for \( B \), then this schedule will again be serializable as the values read and written would be equivalent to the serial execution.

\[
\begin{array}{llll}
\text{Start}(TX1) & \text{Start}(TX2) & \text{Start}(TX3) \\
\text{Read}(TX1,A) & \text{Write}(TX2,A) & \text{Write}(TX2,B) \\
\text{...} & \text{Write}(TX2,B) & \text{Commit}(TX2) \\
\text{Write}(TX1,B) & \text{Commit}(TX2) & \text{...} \\
\text{Commit}(TX1) & \text{Commit}(TX2) & \text{Write}(TX3,B) \\
\text{Commit}(TX1) & \text{Commit}(TX2) & \text{Commit}(TX3) \\
\end{array}
\]

(a) A serializable schedule.

\[
\begin{array}{llll}
\text{Start}(TX3) & \text{Start}(TX2) & \text{Start}(TX1) \\
\text{Write}(TX3,B) & \text{Write}(TX2,A) & \text{Write}(TX1,A) \\
\text{Commit}(TX3) & \text{Write}(TX2,B) & \text{...} \\
\text{Write}(TX1,B) & \text{Commit}(TX2) & \text{...} \\
\text{Commit}(TX1) & \text{Commit}(TX2) & \text{Commit}(TX3) \\
\end{array}
\]

(b) A non-serializable schedule.

(c) The serializability graph of the non-serializable schedule.

Figure 3.2: Serializable and non-serializable schedules.

Transactional actions often result in non-serial schedules due to the concurrent execution of transactions. Thus, TM systems implement concurrency control algorithms (or simply concurrency algorithms) to only accept serializable (in addition to serial) schedules and to avoid non-serializable schedules in order to preserve consistency. For this purpose, TM systems keep track of transactional actions in order to verify that the serializability property holds. The con-
currency control algorithms are classified into two classes: optimistic concurrency algorithms and pessimistic concurrency algorithms. An optimistic concurrency algorithm allows transactions to perform their actions without any checks, and verifies that serializability holds when a transaction attempts to commit [9]. A pessimistic concurrency algorithm does not allow actions that may potentially result in non-serializable execution [9].

The next section analyzes the traditional concurrency algorithm implemented by the existing TM systems.

### 3.4 Concurrency Control in Transactional Memory

TM systems (both hardware proposals and software implementations) emulate the two-phase-locking (2PL) concurrency algorithm implemented by the database management systems (DBMSs). In this database algorithm, a database transaction must acquire a distinct exclusive lock before reading or writing to each database element and release these locks only when it is ready to commit [9]. Thus, conflicting accesses to shared elements are not allowed from the first time the element is accessed until the commit time.

TM systems also implement the 2PL concurrency algorithm by not allowing conflicting accesses: every single conflict either causes a transaction to abort or the conflicting access to be delayed until the conflicting transaction commits or aborts. As described in Chapter 2, TM design space consists of a large spectrum of policies on conflict detection and version management. Thus, the design choices of a particular TM implementation impact how 2PL is implemented. For instance, EL and EE HTM systems abort a transaction as soon as a conflicting accesses is issued. In contrast, LL HTM systems allow the accesses to be issued, but abort conflicting transactions at commit time. Blocking STM implementations implement 2PL explicitly by acquiring locks for each shared item accessed by a transaction and by holding these locks until the transaction commits. Obstruction-free STM systems abort transactions when conflicts are detected either at access time, at commit time, or during read-validation.
Figure 3.3 summarizes how 2PL is implemented by different classes of TM systems.

<table>
<thead>
<tr>
<th>TM</th>
<th>TM</th>
<th>HTM</th>
<th>EE</th>
<th>A transaction is aborted when a conflicting read or write access is issued.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>EL</td>
<td>A transaction is aborted when a conflicting read or write access is issued.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>LL</td>
<td>A transaction is aborted when a conflicting transaction commits.</td>
</tr>
<tr>
<td></td>
<td>Obstruction</td>
<td>Eager-acquire</td>
<td></td>
<td>A transaction is aborted when an item is accessed in read or write mode in a conflicting way.</td>
</tr>
<tr>
<td></td>
<td>-free Systems</td>
<td>writes,</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>visible reads</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Eager-acquire</td>
<td></td>
<td>A transaction is aborted when an item is accessed in write mode which conflicts with an earlier write; or read-validation detects that an item accessed in read mode is also opened in write mode by another transaction.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>writes,</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>invisible reads</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Lazy-acquire</td>
<td></td>
<td>A transaction is aborted when an item accessed in write mode is simultaneously committed by more than one transactions; or when an item opened in read mode is committed by another transaction.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>writes,</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>visible reads</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Lazy-acquire</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>writes,</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>invisible reads</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Blocking Systems</td>
<td></td>
<td></td>
<td>A transaction is blocked when it tries to access an item in a conflicting way until the transaction that accessed the item earlier commits or aborts.</td>
</tr>
</tbody>
</table>

Figure 3.3: 2PL implementation of TM systems.

The 2PL concurrency algorithm is illustrated using the example in Figure 3.4(a). The figure shows the schedule of three transactions TX1, TX2 and TX3 that accesses shared items A and B. In the schedule, TX1 and TX2 conflict over B, and TX1 and TX3 conflict over A. Thus, TM systems today will abort at least one of the transactions (depending on the conflict management algorithm they implement), or delay the conflicting accesses until one the conflicting transactions commit. For instance, an EL HTM system will abort TX1 and TX2 when `Write(TX3,A)` is issued, in accordance with the requester-wins policy. On the other hand, a blocking STM system will delay `Write(TX2,B)` until TX1 commits, and will delay
Write(TX3,A) until TX2 commits.

The 2PL concurrency algorithm results in consistent state and correct execution because it guarantees serializability [9]. This is because in every schedule that is valid under 2PL the conflicting actions of transactions are non-interleaving; only non-conflicting actions can be interleaving. Therefore, we can transform a non-serial schedule that is valid under 2PL (i.e. a schedule where non-conflicting actions are interleaving) to an equivalent serial schedule by just re-ordering the non-conflicting actions of transactions. The order of the conflicting actions (i.e. non-interleaving actions) remains the same and hence the transactions read and write the same values of shared data in the non-serial and in its equivalent serial schedules.

However, 2PL has also been shown to be a strict algorithm of serializability for database transactions [37]. That is, for a fixed number of actions performed by a fixed number of transactions, the set of serializable schedules is a superset of the set of schedules that obey 2PL. Therefore, while TM systems guarantee serializability by using 2PL, it is not necessary for serializability. This can be seen in Figure 3.4(a) which shows a schedule that is not valid under 2PL. However, this schedule is in fact serializable; TX1→TX2→TX3 is a valid serialization. Consequently, all three transactions can commit successfully despite their conflicts without any aborts or delays.

Thus, the use of 2PL in TM systems results in lack of concurrency and/or unnecessary aborts. This is because some schedules that do not obey 2PL may very well be serializable and thus produce correct outcome and consistent state. For applications that have abundant parallelism and low contention, the use of 2PL is not likely to be a problem because for such applications abort rates are low to start with. Hence, the impact of the aborted transactions on performance is minimal. This is evident by the good performance of such applications on many STM and HTM systems [12, 15, 22, 31].

In contrast, applications with long transactions and high contention do not perform well on today’s TM systems. Our hypothesis is that such applications perform poorly is because 2PL is overly conservative. 2PL limits concurrency by causing transactions to stall or to abort when
they could be successfully committed, which degrades performance.

We recognize that 2PL enforces a special type of serializability (described below) and that other concurrency control algorithms can also be used to enforce this type of serializability, which can results in correct TM implementations. Thus, we propose to improve the performance of applications with long transactions and high contention through the use of a more relaxed concurrency algorithm instead of 2PL.

The following section describes the special type of serializability enforced by the TM systems.

### 3.5 Conflict Serializability

Two schedules are said to be *conflict-equivalent* if (1) they contain exact same actions and (2) the order of the conflicting actions in one schedule is the same as the other. A schedule is *conflict-serializable* if it is conflict-equivalent to one or more serial schedules. Whether a schedule holds the conflict-serializability (CS) property can be tested using a *serializability graph* [9]. In this graph, each node represents a transaction and each edge represents a conflicting action between two transactions. The edges are directed and are incident from the transaction that performs its action earlier in time. A schedule is conflict-serializable if its serializability graph is acyclic [9].

For instance, the schedule shown in Figure 3.4(a) is conflict-serializable. The serializability graph shown in Figure 3.4(b) contains three nodes for each transaction TX1, TX2 and TX3. There are two edges between these nodes caused by the conflicting accesses of the transactions: an edge incident from TX1 to TX2 caused by the conflicting accesses over B, and another edge incident from TX2 to TX3 caused by the conflicting accesses over A. Thus, the serializability graph is acyclic and gives us the valid serialization TX1→TX2→TX3. In contrast, the schedule in Figure 3.2(b) is not conflict-serializable due to the conflicting accesses of transactions; TX1 reads A before TX2 writes to it, and TX1 writes to B after TX2 writes to it. Thus, the
order of these two conflicting accesses are opposite to each other, which causes a cycle in the serializability graph of the schedule which is shown in Figure 3.2(c).

![Schedule Diagram](image)

(a) The schedule.

![Serializability Graph](image)

(b) The serializability graph.

Figure 3.4: 2PL concurrency algorithm causes aborts and/or delays.

The CS model always produces correct execution and consistent state because any schedule that is conflict-serializable is serializable [9]. This is because a valid serialization for transactions can be obtained using breadth-first traversal of the acyclic serializability graph and the non-serial conflict-serializable schedule can be transformed to this equivalent serial schedule by only changing the order of the non-conflicting actions. However, a schedule that is serializable is not necessarily conflict-serializable. This can be seen in Figure 3.2(a) which shows a schedule which is serializable but not conflict-serializable.

2PL is a strict implementation of the CS model, i.e., for a fixed set of transactional actions, the set of conflict-equivalent schedules is a superset of the set of transactions that are valid under 2PL [9, 37]. This is because (1) any schedule that is valid under 2PL is also valid under CS, (2) any schedule that is not valid under CS is also not valid under 2PL, and (3) there exist

---

1The reason why this schedule is serializable is due to another type of serializability called view-serializability [9] which is outside the scope of this thesis.
schedules that are conflict-serializable but invalid under 2PL. The first statement is true because any schedule that is valid under 2PL does not contain any conflicting actions between active transactions; therefore the serializability graph of such a schedule will contain no edges, and hence, will be acyclic. The second statement is true because a schedule that is not valid under CS contains conflicts between transactions, and hence, is invalid under 2PL. Finally, the third statement is true because a schedule that contains any conflicts is invalid under 2PL, yet it can be conflict-serializable if the order of the conflicting actions is the same for all transactions. There exists at least one schedule shown in Figure 3.4(a) which is conflict-serializable but invalid under 2PL.

3.6 Implications for TM Systems

We recognize that CS can also be enforced through the use of more relaxed concurrency control algorithms in TM systems. Such algorithms can allow interleaving of conflicting actions and, hence, conflicting transactions to commit successfully in some cases. We hypothesize that compared to a TM system that implements 2PL a TM system that implements such a more relaxed algorithm can incur fewer aborts or delays. We believe that this can improve concurrency and hence performance in TM systems. Such improvement in performance would be more pronounced for applications that contain long transactions and high contention. This is because for such applications aborts are more frequent. Thus the penalty for re-executing aborted transactions is higher.

However, despite its benefits, using more relaxed algorithms to enforce CS can be more complex than using 2PL. This is because CS requires keeping track of the order of all the conflicting actions in order to verify that transactions can be serialized. Thus, straightforward implementations of CS can incur high overheads and slow down transactional actions. For instance, implementing CS using a global serializability graph is not a feasible approach because this requires searching for cycles in the graph after every transactional action. Therefore, we
propose a simple algorithm that can be used to efficiently implement CS in TM systems. In
the remainder of this thesis, we describe this algorithm, show how we implement it in software
and in a simulator of an HTM system, and present the performance of these implementations.
Chapter 4

Serializability Order Numbers

We propose an algorithm for efficiently implementing CS in TM systems. We refer to this algorithm as the \textit{the SON algorithm}. This chapter gives an overview of the algorithm, formulates the algorithm of the method, shows that it produces serializable schedules, analyzes the safety, the classification and the limitations of this algorithm, and illustrates how the algorithm works using examples.

4.1 Overview

The algorithm is based on the fact that if a conflict-equivalent serial schedule can be constructed, then the transactions are serializable. Thus, the algorithm attempts to incrementally construct a conflict-equivalent serial schedule. This conflict-equivalent serial schedule is constructed by determining a \textit{serializability order number} (SON) for each transaction. SONs indicate the relative order of all transactions in the conflict-equivalent serial schedule that is being constructed. That is, a transaction that has a smaller SON is serialized earlier than a transaction that has a larger SON. As described in Section 3.5, the order of the transactions in the conflict-equivalent serial schedule is determined by the relative order of their conflicting actions. Thus, the SONs of transactions are also determined based on the relative order of their conflicting actions. That is, a transaction that performs a conflicting access earlier will have a smaller SON.
than another transaction that performs the conflicting access later. If a SON can be determined for each transaction, then a conflict-equivalent serial schedule exists. Hence, effectively, it can assumed that all the read and write actions of transactions happen atomically and in the order of their SONs. Otherwise, if a unique SON cannot be determined for a transaction, the method cannot guarantee that this transaction can be serialized with respect to the other transactions that already committed. Thus, the transaction aborts and retries.

Transactions are assigned SONs when they commit. Thus, two conflicting transactions are serialized only when one of these transactions commits and is assigned an SON. More specifically, among two transactions TX\textsubscript{a} and TX\textsubscript{c} that conflict over a shared item \(i\), the still active transaction TX\textsubscript{a} is serialized with respect to the other transaction TX\textsubscript{c} that commits earlier using the following rules.

**Forward-Serialization.** If TX\textsubscript{a} accesses (reads or writes to) \(i\) after TX\textsubscript{c} does, then TX\textsubscript{a} must also be serialized after TX\textsubscript{c}. Thus, the SON of TX\textsubscript{a} must be higher than that of TX\textsubscript{c}. We say that the active transaction TX\textsubscript{a} is forward-serialized after the committed transaction TX\textsubscript{c}.

**Backward-Serialization.** If TX\textsubscript{a} accesses \(i\) before TX\textsubscript{c} does, then TX\textsubscript{a} must also be serialized before TX\textsubscript{c}. Thus, TX\textsubscript{a}'s SON must be lower than that of TX\textsubscript{c}. We say that the active transaction TX\textsubscript{a} is backward-serialized before the committed transaction TX\textsubscript{c}.

For instance, in the schedule shown in Figure 3.4(a), TX\textsubscript{2} can be backward-serialized before TX\textsubscript{3} and TX\textsubscript{1} can be backward-serialized before TX\textsubscript{2}. Thus, all the transactions can be assigned SONs based on the order of their conflicting actions. For instance, TX\textsubscript{1} can receive an SON of 1, TX\textsubscript{2} can receive an SON of 2, and TX\textsubscript{3} can receive an SON of 3. The order of these SONs matches the order their conflicting actions; TX\textsubscript{2} writes to \(B\) after TX\textsubscript{1} reads it, and TX\textsubscript{3} writes to \(A\) after TX\textsubscript{1} and TX\textsubscript{2} read it.

In contrast, in the schedule shown in Figure 3.2(b), TX\textsubscript{1} cannot be assigned a unique SON. This is because it must be backward-serialized before TX\textsubscript{2} (i.e., its SON must be lower than that of TX\textsubscript{2}) because it reads \(A\) before TX\textsubscript{2} writes to it, and it must also be forward-serialized after TX\textsubscript{2} (i.e., its SON must also be higher than that of TX\textsubscript{2}) because it writes to \(B\) after
TX2 writes to it. Thus, TX1 must be aborted. This is consistent with the conclusion that the schedule is not conflict-serializable, which is drawn from the serializability graph shown in Figure 3.2(c).

The serialization rules impose a lower bound (LB) and an upper bound (UB) on the SON of a transaction. In our algorithm, the lower bound of a transaction is initialized to 0, and the forward-serialization rule is applied to increase it. The upper bound is initialized to $\infty$ and the backward-serialization rule is used to lower it. If at any moment during execution, the lower bound on the SON of a transaction becomes equal to or higher than its upper bound, this transaction cannot be placed in a conflict-equivalent serial schedule. In this case, the transaction aborts.

If a transaction performs all its accesses without aborting, it starts to commit. Then, it examines its SON range to determine a specific SON value. If the upper bound of the range is not $\infty$, then it reflects the SON of some conflicting transaction. Therefore, the SON of the transaction is selected as the upper bound minus one. If the upper bound is $\infty$, then the SON of the transaction is set to the lower bound plus $n$, where $n$ is the number of threads. In both cases, the choice of the SON reflects the need to assign a sufficiently high SON to each transaction so that it is possible to assign unique smaller SON values to the remaining $n - 1$ active transactions. Selecting smaller SONs may cause some conflicting transactions to abort unnecessarily because they cannot be assigned unique SONs. Selecting larger SONs, on the other hand, may cause the SON values to overflow. The consequences of overflowed SON values is discussed later in this chapter (Section 4.5).

4.2 Formulation

In order to dynamically generate a conflict-equivalent serial schedule, serializations between transactions must be created for each conflict, depending on the order of the conflicting accesses. Conflicts can be of three types: read-after-write (RAW), write-after-write (WAW) and
write-after-read (WAR).

For an active transaction $TX_i$ and a data item $di$, $RS(TX_i)$ denotes the read-set of $TX_i$ and $WS(TX_i)$ denotes the write-set of $TX_i$. $SON(TX_i)$ denotes the SON of transaction $TX_i$, $LB(TX_i)$ denotes the SON lower bound and $UB(TX_i)$ denotes the SON upper bound of $TX_i$.

Below we formulate the operations on $LB(TX_i)$, $UB(TX_i)$, and $SON(TX_i)$ during the execution of $TX_i$, for each type of conflict.

Note that the isolation property of TM guarantees that writes are invisible until transactions commit. Thus, if a write action causes a conflict, then the transaction that performed the write action must have committed and, hence, is already assigned an SON. In contrast, read actions can cause conflicts before transactions commit.

1. Transaction Start:

   \[
   LB(TX_i) = 0 \\
   UB(TX_i) = \infty
   \]

2. Transaction Validation:

   \[
   \text{if } LB(TX_i) \geq UB(TX_i) \\
   \text{Abort}(TX_i)
   \]

3. Transaction Serialization:

   \[
   \text{if } UB(TX_i) = \infty \\
   \text{SON}(TX_i) = LB(TX_i) + \text{num\_threads}; \\
   \text{else} \\
   \text{SON}(TX_i) = UB(TX_i) - 1;
   \]

4. RAW Conflict caused by $\text{Read}(TX_i, di)$:

   \[
   \text{for each transaction } TX_w \\
   \text{if } di \in WS(TX_w) \\
   LB(TX_i) = \max(LB(TX_i), SON(TX_w))
   \]

5. WAW Conflict caused by $\text{Write}(TX_i, di)$:

   \[
   \text{for each transaction } TX_w \\
   \text{if } di \in WS(TX_w) \\
   LB(TX_i) = \max(LB(TX_i), SON(TX_w))
   \]
6. WAR Conflict caused by Write(TXi, di):

   for each transaction TXr
   if di ∈ RS(TXr)
     if TXr has committed
       LB(TXi) = max(LB(TXi), SON(TXr))
     else if TXr is active
       UB(TXr) = min(UB(TXr), SON(TXr))

Figure 4.1 shows all nine possible conflicting schedules for read and write actions of two transactions in a TM system that uses lazy VM (i.e., writes take effect at commit). Section 4.6 discusses the impact of using other TM techniques (in particular eager VM) to the implementation of the SON method.

The figure shows what type of conflicts exist in each schedule, how transactions must be serialized and how this serialization is achieved using SONs. It can be seen that the SON algorithm properly serializes transactions according to the order of their conflicting actions in every possible ordering of read, write and commit actions of transactions.

### 4.3 Read-Numbers and Write-Numbers

The SON algorithm presented above finds conflicting accesses with any past transaction that ever executed. For this purpose, the read and write sets of every past transaction have to be kept and possible conflicts with every past transaction have to be searched for at every transactional action. These can incur significant overheads in a TM system. Therefore, a simpler technique can be used for more efficient implementation. More specifically, when a transaction commits, it assigns its SON to all the data items it read or wrote to. The largest SON of all the transactions that wrote to a data item is referred to as the write-number of the data item. Similarly, the largest SON of all the transactions that read a data item is referred to as the read-number of the data item.

When a transaction reads or writes to a data item, its lower bound is updated with the write-number of the data item, effectively serializing the transaction after all the transactions that
### Figure 4.1: All nine possible schedules for conflicting read and write actions under lazy VM.

<table>
<thead>
<tr>
<th>Schedule #</th>
<th>Action Order</th>
<th>Read-After-Write</th>
<th>Write-After-Write</th>
<th>Write-After-Read</th>
</tr>
</thead>
<tbody>
<tr>
<td>Schedule</td>
<td>Start(TX1)</td>
<td>Start(TX2)</td>
<td>Start(TX1)</td>
<td>Start(TX2)</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>Write(TX1,A)</td>
<td>...</td>
<td>Write(TX1,A)</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>Commit(TX1)</td>
<td>Read(TX2,A)</td>
<td>Commit(TX1)</td>
<td>Write(TX2,A)</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>Commit(TX2)</td>
<td></td>
<td></td>
<td>Commit(TX2)</td>
</tr>
<tr>
<td>Conflict Type</td>
<td>RAW</td>
<td>RAW</td>
<td>RAW</td>
<td>RAW</td>
</tr>
<tr>
<td>Serialization</td>
<td>TX1 → TX2</td>
<td>TX1 → TX2</td>
<td>TX1 → TX2</td>
<td>TX1 → TX2</td>
</tr>
<tr>
<td>Operation</td>
<td>LB(TX1) = SON(TX1)</td>
<td>LB(TX2) = SON(TX1)</td>
<td>LB(TX2) = SON(TX1)</td>
<td>LB(TX2) = SON(TX1)</td>
</tr>
<tr>
<td>Schedule #</td>
<td>Start(TX1)</td>
<td>Start(TX2)</td>
<td>Start(TX1)</td>
<td>Start(TX2)</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>Write(TX1,A)</td>
<td>...</td>
<td>Write(TX1,A)</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>Commit(TX1)</td>
<td>Read(TX2,A)</td>
<td>Commit(TX1)</td>
<td>Write(TX2,A)</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>Commit(TX2)</td>
<td></td>
<td></td>
<td>Commit(TX2)</td>
</tr>
<tr>
<td>Conflict Type</td>
<td>WAR</td>
<td>WAR</td>
<td>WAR</td>
<td>WAR</td>
</tr>
<tr>
<td>Serialization</td>
<td>TX2 → TX1</td>
<td>TX2 → TX1</td>
<td>TX2 → TX1</td>
<td>TX1 → TX2</td>
</tr>
<tr>
<td>Operation</td>
<td>LB(TX1) = SON(TX2)</td>
<td>LB(TX1) = SON(TX2)</td>
<td>LB(TX1) = SON(TX2)</td>
<td>UB(TX1) = SON(TX2)</td>
</tr>
<tr>
<td>Schedule #</td>
<td>Start(TX1)</td>
<td>Start(TX2)</td>
<td>Start(TX1)</td>
<td>Start(TX2)</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>Write(TX1,A)</td>
<td>...</td>
<td>Write(TX1,A)</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>Commit(TX1)</td>
<td>Read(TX2,A)</td>
<td>Commit(TX1)</td>
<td>Write(TX2,A)</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>Commit(TX2)</td>
<td></td>
<td></td>
<td>Commit(TX2)</td>
</tr>
<tr>
<td>Conflict Type</td>
<td>WAR</td>
<td>WAR</td>
<td>WAR</td>
<td>WAR</td>
</tr>
<tr>
<td>Serialization</td>
<td>TX2 → TX1</td>
<td>TX1 → TX2</td>
<td>TX1 → TX2</td>
<td>TX1 → TX2</td>
</tr>
<tr>
<td>Operation</td>
<td>UB(TX2) = SON(TX1)</td>
<td>LB(TX2) = SON(TX1)</td>
<td>LB(TX2) = SON(TX1)</td>
<td>LB(TX2) = SON(TX1)</td>
</tr>
</tbody>
</table>
wrote to the address in the past. Similarly, when a transaction writes to a data item, its lower bound is updated with both the read-number and the write-number, effectively serializing the transaction after all the past readers and writers of the data item. However, even though write-numbers and read-numbers are used to serialize committed transactions, the upper bound of an active reader transaction still needs to be updated with the SON of a writer transaction if a WAR conflict is detected. Thus, the read and write actions are formulated as shown below, where \( \text{WN}(\text{di}) \) denotes the write-number of a data item \( \text{di} \) and \( \text{RN}(\text{di}) \) denotes the read-number of \( \text{di} \).

\[
\begin{align*}
\text{Read}(\text{TX}_i,\text{di}): \\
\quad \text{LB}(\text{TX}_i) &= \max(\text{LB}(\text{TX}_i), \text{WN}(\text{di})) \\
\text{Write}(\text{TX}_i,\text{di}): \\
\quad \text{LB}(\text{TX}_i) &= \max(\text{LB}(\text{TX}_i), \text{WN}(\text{di})) \\
\quad \text{LB}(\text{TX}_i) &= \max(\text{LB}(\text{TX}_i), \text{RN}(\text{di})) \\
\quad \text{for each transaction } \text{TX}_r \\
\quad \quad \text{if } \text{di} \in \text{RS}(\text{TX}_r) \\
\quad \quad \quad \text{if } \text{TX}_r \text{ is active} \\
\quad \quad \quad \quad \text{UB}(\text{TX}_r) &= \min(\text{UB}(\text{TX}_r), \text{SON}(\text{TX}_i))
\end{align*}
\]

### 4.4 Safety Analysis

In order to prove that the SON technique produces conflict-serializable schedules, we need to show that a cycle in the serializability graph of a schedule causes invalid SON bounds, and therefore is safe.

First, let’s assume two transactions \( \text{TX}_1 \) and \( \text{TX}_2 \) and an edge \( \text{TX}_1 \rightarrow \text{TX}_2 \) in the serializability graph of the schedule. This edge is caused by a conflict between \( \text{TX}_1 \) and \( \text{TX}_2 \) and therefore this will result in \( \text{TX}_1 \)’s SON to be lower than \( \text{TX}_2 \)’s SON, i.e., \( \text{SON}(\text{TX}_1) < \text{SON}(\text{TX}_2) \). This is ensured by the forward-serialization and backward-serialization rules. Now, let’s assume a cycle in the serializability graph which consists of transactions \( \text{TX}_1, \text{TX}_2, \text{TX}_3, \ldots, \text{TX}_N, \) and \( \text{TX}_1 \). Consequently, we have \( \text{SON}(\text{TX}_1) < \text{SON}(\text{TX}_2), \text{SON}(\text{TX}_3) < \)
... < SON(TXN) < SON(TX1). This gives SON(TX1) < SON(TX1) which is a contradiction and will cause one of the transactions to have an invalid SON bound and hence to abort. Thus, the technique will always produce serializable schedules.

### 4.5 Classification and limitations

In this section, we discuss the classification and the limitations of the SON method. That is, we discuss how the set of schedules valid under the SON method is a subset of the conflict-serializable schedules and a superset of the schedules valid under 2PL. We also discuss how overflowed SONs impact the correctness and the performance.

**Proposition 1.** The SON technique is an approximation of CS and therefore is a subclass of CS, i.e., transactions may abort in the absence of a cycle in the serializability graph.

Let’s assume a serialization \( TXa1 \rightarrow TXa2 \rightarrow \ldots \rightarrow TXaN \) caused by conflicting accesses to a data item \( a \). The SON algorithm will assign to these transactions SONs that increase monotonically starting from zero. Now, let’s assume another serialization of different transactions \( TXb1 \rightarrow TXb2 \rightarrow \ldots \rightarrow TXbM \) caused by conflicting accesses to another data item \( b \). Consequently, the SON values of these transactions will also monotonically increase starting from zero. Even though the transactions accessing the data item \( a \) and the transactions accessing the data item \( b \) do not conflict over these two data items, the assignment of SON values imposes an ordering among these transactions, i.e., \( SON(TXa1) = SON(TXb1) < SON(TXa2) = SON(TXb2) \). Thus, the assignment of SONs may impose unnecessary constraints. For example, let’s assume a conflict between \( TXa1 \) and \( TXb2 \) over accesses to a data item \( c \) which corresponds to an edge \( TXb2 \rightarrow TXa1 \) in the serializability graph. Now, we have the constraint \( SON(TXb2) < SON(TXa1) \) which contradicts with the earlier constraint, thus causes one of the transactions to abort.

**Proposition 2.** The SON algorithm is more relaxed than the 2PL concurrency control algorithm. That is, the set of schedules valid under the SON algorithm is a superset of the class of
We analyze the behaviour of both algorithms for all types of conflicts and show that 2PL results in more aborts. First, in the absence of WAR and WAW conflicts with active transactions, 2PL does not cause any aborts. This is also valid for the SON algorithm because in the absence of WAR conflicts upper bounds of transactions will not be updated (WAW conflicts do not require upper bound updates). This causes every transaction to have valid SON bounds and thus to commit successfully. Second, 2PL aborts a transaction every time a WAR or a WAW conflict occurs. However, this is not the case for the SON algorithm because these conflicts only cause updates on the lower and upper bounds of transactions. A conflicting transaction will abort only if the update leads to invalid bounds. Third, the SON algorithm updates transaction lower bounds for RAW conflicts that 2PL ignores in a lazy VM because this type of conflict only happens after the writer transaction commits. In contrast, in the SON algorithm, only lower bounds of transactions are updated for RAW conflicts which again does not lead to aborts by itself in the absence of WAR conflicts.

For instance, the schedule shown in Figure 4.2(a) is not valid under 2PL, hence transactions cannot commit successfully without aborts or delays. This is because $TX3$ writes to $A$ after $TX1$ and $TX2$ read it, while both transactions are still active. In contrast, as it will discussed later, the SON algorithm allows all three transactions to complete successfully.

**Proposition 3.** The order of SONs does not necessarily match the commit order of transactions. That is, a transaction that commits after another may obtain a lower SON. This is because SONs reflect the order of transactions in the equivalent serial schedule. This order is determined only by the conflicts between transactions but not by the commit order of transactions. Thus, if two transactions do not conflict, they can be placed in the equivalent serial schedule in any order.

**Proposition 4.** SON bounds can overflow.

SONs are represented as integers in a typical TM system as this results in simpler implementation. However, SONs can overflow if the total number of transactions executed exceeds a
certain threshold. When this happens, the SONs values can wrap around the maximum integer value (i.e., INT_MAX) and the new transactions will obtain lower SON values starting from zero, giving the impression that they commit in the past. This may cause some transactions to abort unnecessarily if their SON upper bounds are updated with overflowed SONs. However, the correctness of execution will not be affected. This is because transactions that have no conflicts commit successfully irrespective of their SON values because their SON upper bound will be $\infty$. Thus, SON overflow causes transactions with conflicts to abort conservatively and no transaction that should abort is allowed to commit successfully. In the extreme, every upper bound update will cause a transaction to abort, imitating the 2PL algorithm.

A stop-the-world technique can be used when the write-numbers and the read-numbers overflow. With this technique, the TM system stops starting any new transactions. When there are no active transactions left running, it re-initializes all the write-numbers and the read-numbers, and resumes the execution by starting the new transactions. Because there are no active transactions running when the execution stops, the transactions that commit earlier are effectively serialized before the transactions that start after. Thus, clearing the write-numbers and the read-numbers have no effect on the serializations.

Using signed numbers to represent the SON bounds can provide an easy way of detecting the overflows. Once a negative SON value is detected (which represents an overflowed SON), the system can take the appropriate measures discussed above.

### 4.6 Implementation Requirements

The SON method described above can more easily be used with lazy VM (or write-buffering in software) than with eager VM (or undo-logging in software). This is because transactions are not assigned SONs until commit time and, hence, active transactions cannot be serialized using SONs immediately when they conflict. Therefore, active transactions cannot be guaranteed to be serializable when conflicts are present. Thus, with eager VM, transactions would be allowed
to read data written by active transactions that are possibly non-serializable. This may cause side effects such as infinite loops, segmentation faults, data corruption, etc. Further, this will also lead to cascading aborts, where aborting a transaction requires aborting all the transactions that read data written by that aborted transaction.

Lazy VM, on the other hand, keeps transactions isolated until commit time when they are known to be serializable, hence, it does not cause side-effects. Further, transactions are properly serialized according to their conflicts at commit time when the SONs of transactions are known. Thus, we analyze and evaluate the SON method only for lazy VM systems. That is, (1) transactions can read data written by only committed transactions, hence, a read action cannot cause a conflict with an active transaction, (2) writes only cause conflicts during commit, when the updates are actually committed into the memory, hence, the type of the conflict does not necessary match the order of the transactional actions.

Other than using lazy VM, the SON method does not mandate any specific TM implementation. That is, visible reads or invisible reads (in STMs), object-level conflict detection or word-level conflict-detection (in STMs), eager CD or lazy CD (in HTMs) can safely be used when implementing the method in a TM system. Similar to 2PL implementations, such design decisions would only impact the performance, but not the correctness.

### 4.7 Examples

Figure 4.2 demonstrates the SON algorithm with two example schedules. The write-numbers of data items are shown in italics on the left, and the SON bounds of transactions are shown in bold next to the actions of transactions and are denoted by $\text{SONB}$. The data items are not accessed by any other transaction before the transactions in the figures started, hence the write-numbers are initially all zero. The read-numbers, on the other hand, are not shown as they are not needed in these particular schedules due to the lack of WAR conflicts with committed readers. Further, because a lazy VM system is assumed, the write actions in the figures represent the actual
## Chapter 4. Serializability Order Numbers

<table>
<thead>
<tr>
<th>WN(A)</th>
<th>WN(B)</th>
<th>WN(C)</th>
<th>TX1 Actions</th>
<th>SONS(TX1)</th>
<th>TX2 Actions</th>
<th>SONS(TX2)</th>
<th>TX3 Actions</th>
<th>SONS(TX3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Start(TX1)</td>
<td>((1,\infty))</td>
<td>Start(TX2)</td>
<td>((0,\infty))</td>
<td>Start(TX3)</td>
<td>((0,\infty))</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Read(TX1,A)</td>
<td></td>
<td>Read(TX2,A)</td>
<td></td>
<td>Read(TX3,A)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>((0,3))</td>
<td></td>
<td>((0,3))</td>
<td>Commit(TX3)</td>
<td>SON=3</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Read(TX1,B)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>0</td>
<td>((0,2))</td>
<td>Commit(TX2)</td>
<td>SON=2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Write(TX1,C)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Write(TX1,C)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
<td>Commit(TX1)</td>
<td>SON=1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(a) All transactions serialize and commit successfully.

<table>
<thead>
<tr>
<th>WN(A)</th>
<th>WN(B)</th>
<th>TX1 Actions</th>
<th>SONS(TX1)</th>
<th>TX2 Actions</th>
<th>SONS(TX2)</th>
<th>TX3 Actions</th>
<th>SONS(TX3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Start(TX1)</td>
<td>((1,\infty))</td>
<td>Start(TX2)</td>
<td>((0,\infty))</td>
<td>Start(TX3)</td>
<td>((0,\infty))</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Write(TX2,A)</td>
<td></td>
<td>Write(TX2,B)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>((3,\infty))</td>
<td>Commit(TX2)</td>
<td>SON=3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Write(TX3,A)</td>
<td></td>
<td>Write(TX3,B)</td>
<td>((3,\infty))</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>((3,6))</td>
<td>Commit(TX3)</td>
<td>SON=6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Read(TX1,B)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Abort(TX1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(b) TX1 aborts due to inconsistent reads.

Figure 4.2: Example schedules demonstrating the SON algorithm.
moment when the write validation takes place atomically at commit time.

Figure 4.2(a) depicts a schedule which commonly appears in data structure implementations, particularly in linked-lists. In this schedule, there are three transactions that all start at the same time with initial SON bounds of \((0: \infty)\). Because the write-numbers of the data items are all initially zero, the transactional actions do not change the SON bounds of transactions until \(TX3\) commits. When \(TX3\) commits, \(SONB(TX3) = (0: \infty)\) and, hence, \(TX3\) is assigned a SON value of 3, which is \(LB(TX3)\) plus 3, the number of threads. Next, \(TX3\) updates the write-number of \(A\) with its SON. It also updates the SON upper bounds of all other active transactions that read data item \(A\) in order to create serializations due to WAR conflicts. Thus, the upper bounds of \(TX1\) and \(TX2\) are updated to 3. When \(TX2\) commits, \(SONB(TX2) = (0, 3)\). Thus, \(TX2\) is assigned a SON of 2, which is \(UB(TX2)\) minus 1. Next, \(TX2\) updates the write-number of \(B\) with its SON and finds that \(TX1\) has already read the data item \(B\) and is still active. Thus, \(TX1\)'s SON upper bound is updated to 2. Finally, when \(TX1\) commits, \(SONB(TX1) = (0, 2)\), thus it is assigned the SON of 1, and it successfully commits. Therefore, the SON algorithm successfully creates the serialization \(TX1 \rightarrow TX2 \rightarrow TX3\) by keeping track of the SON bounds of transactions and the write-numbers of data items. Consequently, transactions commit successfully even though conflicts occur among active transactions.

Figure 4.2(b) demonstrates how a common non-serializable schedule is detected with the SON algorithm. First, \(TX2\) writes to \(A\) and \(B\), and it is assigned the SON of 3, i.e., its lower bound plus the number of threads. Consequently, the write-numbers of \(A\) and \(B\) are updated to 3, signifying that \(TX2\) was the latest transaction that wrote to these items. Second, \(TX1\) reads \(A\) which was written by \(TX2\). This causes a RAW conflict with \(TX2\) and requires serializing \(TX1\) after \(TX2\). This is achieved by updating the lower bound of \(TX1\) with the write-number of \(A\). Next, \(TX3\) writes to \(A\) and \(B\). These accesses cause WAW conflicts with \(TX2\), and hence the lower bound of \(TX3\) is updated with the write-numbers of \(A\) and \(B\), serializing \(TX3\) after \(TX2\). When \(TX3\) commits, it receives the SON of 6 and updates the write-numbers of \(A\) and \(B\) with its own SON. The upper bound of \(TX1\) is also updated as \(Write(TX2, A)\) conflicts
with $\text{Read}(TX1, A)$. Next, $TX1$ reads $B$ which causes a RAW conflict with $TX3$, the last transaction that wrote to $B$. Thus, $TX1$ updates its lower bound with the write-number of $B$. This makes the SON bounds of $TX1$ invalid and causes it to abort. This example presents a common non-serializable schedule where writes to $A$ and $B$ are atomic, yet $TX1$ reads $A$ written by $TX2$ and it reads $B$ written by $TX3$. The SON algorithm successfully keeps track of the inconsistent reads of $TX1$ and aborts it.
Chapter 5

Relaxed Concurrency in Software

We implemented the SON algorithm in a STM library, which we refer to as the Toronto Software Transactional Memory (TSTM). The system is intended as a vehicle for exploring the benefits of using CS to enforce serialization.

This chapter first gives an overview of the system, then it describes the programming model, the data structures used, the transactional actions, how race conditions are handled, various optimizations implemented, and finally it discusses some performance considerations.

5.1 TSTM Overview

TSTM is obstruction-free, and it implements both word-level and object-level conflict detection, write-buffering and visible reads. TSTM is obstruction-free because the SON algorithm allows conflicting transactions to continue execution, rather than blocking them. A transaction in TSTM that acquires its write-set at commit time will eventually be aborted by other conflicting transactions if it does not make any progress to complete its commit. Even though TSTM is not as obstruction-free as other systems (such as DSTM [18]) \(^1\), we will continue to refer to TSTM as obstruction-free in the remainder of this thesis.

\(^1\)Transactions do not commit their write-sets atomically, which may cause inconsistencies if transactions die during their commit.
TSTM implements write-buffering because transactions can more easily be serialized using SONs at commit time when they update memory as mentioned in Section 4.6. The visible reads technique is used because it provides an easy way of identifying reader transactions when a writer transaction commits. An implementation that uses invisible reads would have to explicitly update read-numbers at commit time for each data item they read, which may introduce high overheads. Further, the invisible reads technique causes high overheads also because it requires read validation as mentioned in Section 2.2.5.

5.2 Programming Model

A transaction is started by using the \texttt{BEGIN\_TRANSACTION}(tx) keyword, which returns the identifier \texttt{tx} for the current transaction. This transactional identifier is used in transactional actions. The \texttt{END\_TRANSACTION}(tx) keyword completes the transaction \texttt{tx} either by committing it or by aborting it if the commit operation fails.

TSTM implements both a word-based and an object-based programming model. The word-based programming model is used when word-level conflict detection is desired, particularly for applications written in C. The object-based programming model, on the other hand, performs the conflict detection at coarser object-level granularity and it is particularly useful for applications written in C++. In either model, the underlying TM system makes sure that the transactional reads are consistent and the transactional writes are performed in isolation.

The programmer can insert calls to the appropriate TM routines described below to protect accesses to shared data. Alternatively, a compiler can analyze the data accesses and automatically insert these calls [17, 44]. Each call introduces overheads as the TM system keeps a record of accesses and it searches for conflicts. Thus, it is important to use the TM routines only when it is necessary, i.e. when accesses to shared data are made. Failure to use the appropriate TM routines for every shared data access may cause the TM system to miss conflicts and hence result in inconsistent state.
5.2.1 Word-based Programming Model

- `sharedRead(tx, addr)`. It reads from the address `addr` as part of the transaction `tx`. It returns the value of `*addr`.

- `sharedWrite(tx, addr, val)`. It writes the value `val` to the address `addr` as part of the transaction `tx`.

- `sharedReadF(tx, addr)`. It is similar to `sharedRead` but it reads a floating-point typed data from the memory.

- `sharedWriteF(tx, addr, val)`. It is similar to `sharedWrite` but it performs a transactional floating-point write.

5.2.2 Object-based Programming Model

Objects shared among transactions are wrapped with transactional wrappers. In order to access a shared object, the transactional wrapper must be “opened”. The transactional “open” operations are:

- `OpenForRead(tx)`. This operation is used to access a shared object in read-only mode as part of transaction `tx`. It returns a pointer to the shared object or a pointer to the transactional local copy of the object if the object has been opened for write before.

- `OpenForWrite(tx)`. This operation is used to access the shared object in write-only mode as part of transaction `tx`. That is, the data members of the shared object can not be read before being written to first. This operation returns a pointer to a local transactional copy of the shared object.
5.3 Data Structures

The main data structures used in TSTM are a Descriptor class, a SharedItem class, and a Global Readers Table.

5.3.1 Descriptor Class

This class is used to store metadata for transactions. A single Descriptor is created for each thread when TSTM is initialized. \texttt{BEGIN\_TRANSACTION} returns the corresponding transaction for the executing thread. Transaction metadata mainly consists of a unique transaction identification number \texttt{txId}, the write-set, the lower and upper bounds, and the SON.

5.3.2 SharedItem Class

This class is used to represent each shared data item, i.e., either a shared object or a shared word. In the word-based programming model, a certain number of SharedItems are created at initialization and application virtual addresses are hashed into these items. More than one virtual address can map into a single SharedItem due to collisions in the hash function. This only causes false sharing among transactions, thus some superfluous aborts. In the object-based programming model, a SharedItem is allocated every time a shared object is created by the application and it contains a unique identifier number \texttt{sharedId}, a pointer to the wrapped shared object and a pointer to the transactional copy of the wrapped shared object if the object has been opened for write. In addition, regardless of the programming model used, a SharedItem also contains the write-number of the data item and a pointer to the Descriptor of the transaction that owns the object which is used at commit, as it will be discussed in the remainder of this chapter.
5.3.3 Global Readers Table (GRT)

TSTM implements visible reads with the Global Readers Table (GRT) depicted in Figure 5.1(a). In the GRT, each *SharedItem* maps to a single row of the table based on its *sharedId*. The GRT has two columns. The first is the active readers column, which stores for each thread the *txId* of the last transaction that read the corresponding *SharedItem*. The second column is the read-number column which stores the read-number for the corresponding *SharedItem*. The active readers column contains *txIds*. Our choice to use *txIds* instead of pointers to *Descriptors* is driven by the fact that after a particular transaction commits or aborts, the same *Descriptor* is used to store the metadata of the next transaction executed by the same thread. Hence, past transactions can only be uniquely identified using their *txIds*, not with their *Descriptors*.

A transaction is inserted into the GRT when it reads (or opens for read) a *SharedItem*. For this purpose, first the current active reader transaction that belongs to the same thread is fetched. If this transaction has already committed, the read-number column is updated with the SON of this committed transaction. Next, the new reader transaction stores its *txId* to the active readers column, replacing the old (active) reader transaction. Note that, since each thread in TSTM executes a single transaction at a time, the old reader must be either already committed or already aborted or must be the very same transaction that is trying to read the shared item. Thus, a transaction stored in the active readers column is not guaranteed to be active and the transactional actions described in the remainder of this chapter take this into account. Further, active readers are lazily updated. That is, when transactions commit or abort, they do not remove themselves from the GRT. They are replaced only when a new transaction of the same thread reads the same *SharedItem*. Figure 5.1(b) shows the pseudocode for inserting a reader transaction into the GRT.

Because the size of the GRT is limited, more than a single *SharedItem* can map into the same row. This situation manifests itself as false sharing among transactions. That is, two transactions may be inserted as readers in the same row of the table even though they do not
class GlobalReadersTable {
    txID activeReaders[TABLE_SIZE][NUM_THREADS]
    unsigned long readNumber[TABLE_SIZE]
}
InsertToTable(table,tx,shared) {
    sharedID = shared.id
    tID = tx.threadId
    activeTXID = table.activeReaders[sharedID][tID]
    activeTX = getTXFromID(activeTXID)
    if activeTX is COMMITTED
        if activeTX.SON > table.readNumber[sharedID]
            table.readNumber[sharedID] = activeTX.SON
            table.activeReaders[sharedID][tID] = tx.id
}

(b) The pseudocode for insertion.

Figure 5.1: The Global Readers Table.
read the same SharedItem. This does not cause incorrect execution because read-numbers are only updated with larger values. Thus, a writer transaction will always serialize after all the past committed readers. In addition, due to the false sharing, a writer transaction may also serialize after some committed readers that it does not conflict with. However, this only causes some superfluous aborts.

5.4 Transactional Actions

The main challenge in implementing the transactional actions is to serialize transactions according to all the possible conflicting actions, as shown in Figure 4.1. Figure 5.2 shows the pseudocode for the data structures and transactional actions, which are further described in the following sections. Transactional read and transactional write actions are only shown for the word-based programming model. These actions are implemented similarly for the OpenForRead and OpenForWrite operations of the object-based programming model.

Note that the algorithms discussed in this section are subject to race conditions. Thus, they need to be modified to support concurrent transactions. Section 5.5 discusses how race conditions are handled.

Further, the validation aborts transactions when lowerbound $\geq$ upperbound $- 1$. This is unlike the validation stage of the SON algorithm (see Section 4.2) which simply requires lowerbound $\geq$ upperbound. This difference is caused by the fact that in this particular STM implementation the SON lower and upper bounds are implemented as integer values, and hence, lowerbound $\leq$ upperbound $- 2$ is the necessary condition to be able to select an integer SON between the lower and upper bounds.

5.4.1 sharedRead

This action performs the following steps. First, the transaction searches for the address that is being read in its write-set. If the address is found, it reads the local transactional value pre-
viously written. Otherwise, the transaction updates its lower bound with the `writeNumber` of the `SharedItem`. This ensures that the current reader transaction will be serialized in accordance with the RAW conflicts after all the transactions that previously wrote to this shared item and committed successfully (`TX2` of Schedule 1 in Figure 4.1). If the SON bounds of the transaction become invalid, this serialization is impossible, and hence, the reader transaction aborts. Then, the transaction registers itself as a reader of the `SharedItem` by inserting itself to the GRT. Finally, the contents of the address are read from the memory and returned.

5.4.2 `sharedWrite`

This action performs the following steps. First, the transaction updates its lower bound with the `writeNumber` of the `SharedItem`. This serializes the writer transaction after all the current past writers of the shared item (`TX2` of Schedule 2 in Figure 4.1). However, until the transaction actually commits this write into the memory at commit time, other transactions may commit this particular shared item. Thus, the writer transaction will perform another validation for this write at commit time to make sure that it serializes after all the past writers. Finally, the write action completes by buffering the write in its write-set.

5.4.3 `commit`

The commit action consists of four steps:

1. validateWrites. First, the transaction updates its lower bound with the read-number and the write-number of each item in its write-set. This operation serializes the transaction in accordance with the WAW and WAR conflicts with committed readers (`TX2` of Schedule 3, `TX1` of Schedule 4, `TX2` of Schedule 9 in Figure 4.1) and with committed writers (`TX2` of Schedule 2, `TX1` of Schedule 5, `TX2` of Schedule 8 in Figure 4.1) as this was discussed in Section 4.2. After the validation, if the SON bounds are still valid, the transaction prepares for updating the memory, otherwise it aborts. Note that
class Descriptor {
    txID          id
    unsigned long threadId
    unsigned long son
    unsigned long lowerBound
    unsigned long upperBound
    List          writeSet
    stateType     state
}
class SharedItem {
    sharedId      id
    Transaction   owner
    unsigned long writeNumber
}
validate(tx) {
    if tx.upperBound != 0
        if tx.lowerBound >= tx.upperBound-1
            Abort(tx)
}
start(tx) {
    tx.lowerBound = 0
    tx.upperBound = 0
    state = ACTIVE
}
sharedRead(tx,addr) {
    if addr exists in tx.writeSet
        return getValue(tx.writeSet,addr)
    SharedItem sh = MapToItem(addr)
    updateLowerBound(tx,sh.writeNumber)
    InsertToTable(GRT,tx,sh)
    return *addr
}
sharedWrite(tx,addr,val) {
    SharedItem sh = MapToItem(addr)
    updateLowerBound(tx,sh.writeNumber)
    InsertToList(tx.writeSet,sh,addr,val)
}
commit(tx) {
    state = COMMITTING
    validateWrites(tx)
    serialize(tx)
    notifyReaders(tx)
    state = COMMITTED
    commitWrites(tx)
}
updateLowerBound(tx,newlb) {
    tx.lowerBound = max(tx.lowerBound,newlb)
    validate(tx)
}
updateUpperBound(tx,newub) {
    if tx.upperBound == 0
        tx.upperBound = newub
    else
        tx.upperBound = min(tx.upperBound,newub)
    validate(tx)
}
serialize(tx) {
    if tx.upperBound == 0
        tx.son = tx.lowerBound + NUM_THREADS;
    else
        Tx.SON = tx.upperBound - 1;
}
validateWrites(tx) {
    for each sharedItem i in tx.writeSet
        owner = tx
        updateLowerBound(i.writeNumber)
        readnum = GRT.readNumber[i.id]
        updateLowerBound(readnum)
}
notifyReaders(tx) {
    for each sharedItem i in tx.writeSet
        for each thread tid
            readerTXID = GRT.activeReaders[i.id][tid]
            readerTX = getTXFromID(readerTXID)
            if readerTX is ACTIVE
                updateUpperBound(readerTX,tx.son)
            else if readerTX is COMMITTING
                Abort(tx)
            else if readerTX is COMMITTED
                if readerTX.son >= tx.SON
                    Abort(tx)
}
commitWrites(tx) {
    for each sharedItem i in tx.writeSet
        i.writeNumber = max(i.writeNumber,tx.son)
        addr = getAddressForEntry(tx.writeSet,i)
        val = getValueForEntry(tx.writeSet,i)
        *addr = val
        owner = NONE
}

(a) Data structures and actions.
(b) Helper methods for actions.

Figure 5.2: TSTM data structures and transactional actions for sequential execution.
while a transaction is validating, other conflicting transactions may also be committing or performing transactional actions. These situations are subject to race conditions, which are discussed in the remainder of this chapter.

2. **serialize.** The transaction determines its SON within its SON bounds, as described in Section 5.1.

3. **notifyReaders.** The transaction iterates over the active readers of the shared items it wrote to. If a reader is still active, it updates the upper bound of the reader transaction. This creates serializations for the WAR conflicts with the active readers (Schedules 6 and 7 in Figure 4.1). As it was mentioned in earlier, it is possible that the transaction stored as the active reader in the GRT is not active anymore. If the reader transaction is also currently committing, this signifies a serialization race condition between the writer and the reader of the same shared item. We opt to handle this race condition by simply aborting the writer transaction. On the other hand, if the reader transaction has already committed, this signifies a WAR conflict, and hence this reader transaction must serialize before the writer transaction that is currently committing. Hence, the writer transaction verifies that its SON is higher than the SON of the reader transaction. If this is not the case, this signifies a violation of serialization and the writer transaction aborts. This way of handling the race condition admits cases when the writer transaction may have already updated the upper bounds of other reader transactions before it aborts. Thus, it causes a small number of superfluous aborts, especially for short transactions for which the race conditions are more likely to happen because transactions commit more frequently.

4. **commitWrites.** The transaction commits all the shared items in its write-set by updating the write-numbers of the items and by updating the memory contents with the values buffered.
5.5 Race Conditions

In this section, we describe the race conditions that arise due to concurrent execution of transactional actions and how these race conditions are handled.

5.5.1 Committing Writer vs. Committing Writer

This race condition arises when transactions try to simultaneously commit the same shared data item they wrote to. According to the SON algorithm, among two transactions that write to the same data item, the one that commits first gets a lower SON. This prevents the two transactions from validating (i.e., updating their SON bounds) and updating the write-numbers of the data items concurrently. We opt to use a lock-based technique to handle this race condition.

With this technique, when a transaction starts to commit, it first acquires all the shared items in its write-set by setting itself as the owner of the items using a compare-and-swap (CAS) atomic instruction. It releases these items only when it completes its commit. If an item is already acquired, the transaction releases all the items it acquired and aborts in order to avoid deadlocks. This makes sure that validating and updating of transactional writes happen atomically. Although this technique introduces overheads and serializes the commit operations of conflicting transactions, it does not impact the performance significantly because transactions generally write to a small number of shared items and hence a small number of locking and unlocking operations are performed at commit time.

5.5.2 Active Reader vs. Committing Writer

This race condition arises when a shared item is read while another transaction is simultaneously executing a commit action that will commit the same shared item. If the read action executes before the commit (i.e., WAR conflict), the reader transaction must receive a lower SON according to the backward-serialization rule of the SON algorithm. Similarly, if the commit action takes place earlier (i.e., RAW conflict), the reader transaction must receive a higher
SON according to the forward-serialization rule. A serialization violation occurs when (1) the reader transaction updates its lower bound before the committer transaction updates the write-number of the shared item, and (2) the reader transaction inserts itself to the readers table after the committer transaction iterates the table for the readers. As a result, neither the reader transaction nor the writer transaction are aware of the conflict and thus do not update their SON bounds, and hence, transactions are not serialized. This race condition is depicted in Figure 5.3(a).

sharedRead(tx, addr) {
    ... (1) updateLowerBound(tx, item.writeNumber)
    ... (2) notifyReaders(tx)
    ... (3) commitWrites(tx)
    ... (4) InsertToTable(GRT, tx, item)
    ... }

commit(tx) {
    ... 
    ... 
    ... 
    ... }

(a) The ordering of operations that causes a race condition.

sharedRead(tx, addr) {
    if addr exists in tx.writeSet
        return getValueForEntry(tx.writeSet, addr)
    do{
        currentwritenumber = item.writeNumber
        snapshot = *addr
        InsertToTable(GRT, tx, item)
    }while(item.owner != NONE ||
        currentwritenumber != item.writeNumber)
    updateLowerBound(tx, currentwritenumber)
    return snapshot
}

(b) Concurrent sharedRead action.

Figure 5.3: Active reader transaction vs. committing writer race condition.

Since transactions generally read large number of shared items, we opt not to use a lock-based technique to handle this race condition. We rather use a speculative technique where the reader transaction attempts the read action, however, it checks whether another transaction committed the same item in the meantime or is about to commit it. If this is the case, the read operation is repeated. Thus, the sharedRead action proceeds as follows. The reader transaction first records the write-number of the shared item. Second, it takes a snapshot of
the shared item by reading from the memory. The snapshot is guaranteed to be consistent with
the write-number because when an item is committed its write-number is updated before the
memory is updated\(^2\). That is, the write-number loaded by the reader transaction will always
be equal to or higher than the write-number of the transaction that committed the value read
from the memory. Third, the transaction inserts itself as a reader of the item. Then, the reader
transaction compares the current write-number of the item and checks whether the item is
acquired by another transaction. If the current write-number does not match the recorded
write-number or if the item is acquired, this means that the item has either been committed
or is currently being committed by another transaction. In this case, the reader transaction
attempts the read action again by repeating the operations again until the race condition is
avoided. The pseudocode for the concurrent *sharedRead* action is shown in Figure 5.3(b).

### 5.5.3 Committing Reader vs. Committing Writer

As mentioned in Section 5.4, when a transaction is performing *validateWrites* during its
commit, it is possible that it finds a reader transaction in the GRT that is also committing at
the same time. This signifies a serialization race condition between the writer and the reader of
the same shared item. This race condition is handled by simply aborting the committing/writer
transaction.

### 5.6 Performance

There are two main sources of performance overheads associated with the implementation of
the SON algorithm as the concurrency control algorithm.

The first is the algorithmic overhead. This is caused by the higher complexity of the commit
operations (more specifically the *notifyReaders* step) compared to 2PL. Given \(n\) threads

\(^2\)We rely on processor consistency or on memory barriers to guarantee that writes are observed in the program
order.
and $w$ writes in each transaction, the complexity of the commit operation is $O(n \times w)$. This is because during the notifyReaders operation of the commit, the committing transaction iterates over the active readers in the GRT (maximum of $n$ readers assuming a single transaction per thread) of each data item it wrote to (i.e., $w$ item). This $O(n \times w)$ complexity is higher than $O(w)$ complexity of the commit operation under 2PL and visible reads [30].

The second source of overhead arises from the need to maintain active reader transactions and read-numbers in the GRT, and write-numbers for each shared data item. The need to access and update these values results in additional cache misses, hence, increases the latencies of the transactional read and commit actions.

In addition to the performance overheads, maintaining the GRT, write-numbers, and SON lower and upper bounds increase the memory footprint of TSTM. The size of the GRT is a parameter of TSTM and can be adjusted according to the application characteristics. A smaller GRT reduces the memory footprint, but may increase aliasing and lead to unnecessary aborts. When we evaluate the performance TSTM in Section 7.1, we show that a GRT of size 64KB is adequate to obtain good performance across our set of benchmarks.

## 5.7 Optimizations

In this section, we discuss several optimizations performed in order to improve the performance of transactional actions.

### 5.7.1 Retiring Transactions

An important overhead in TSTM is caused by transactions that update the read-numbers in the GRT for the shared items they read. For this purpose, we investigate whether this is necessary for every shared item. We say that a committed transaction is retired if it does not have an impact on the serialization of transactions anymore. Retired transactions can safely be ignored by future transactions. Thus, if a reader transaction stored in the GRT is found to be retired, the
read-number of the shared item is not updated, effectively ignoring the transaction’s impact on
the serialization of future transactions.

A transaction can retire when it is known that it cannot be included in a cycle in the se-
rializability graph in the future. Whether or not a transaction can be included in a cycle can
be determined as follows. When a transaction TXC commits successfully, we know that it is
not part of a cycle. TXC will only be a part of a cycle in the future with the addition of a new
edge in the serializability graph, caused by an access of an active transaction TXA. Since this
access happens after TXC has committed, TXA will be serialized after TXC in the serializability
graph, i.e., TXC→...→TXA. If this new edge creates a cycle in the serializability graph, this
means that TXA is also serialized before TXC, i.e., TXA→...TXC→...→TXA. This is only pos-
sible if TXA was in fact active and serialized before TXC when TXC committed. Thus, we can
conclude that TXC can only be included in a cycle, if there exists a transaction such as TXA.
In other words, TXC cannot anymore be included in a cycle, if at the commit time of TXC,
they were no active transactions that were serialized before TXC. In the SON algorithm, this
condition means that TXC can retire if at commit time there were no active transactions with
an upper bound lower than the SON of TXC. Therefore, finding whether transactions can be
retired is simple and incurs low overheads. However, this simple optimization can improve the
performance significantly by lowering the latencies of transactional read actions, especially in
applications that have long transactions that perform large number of read actions.

5.7.2 The SON Table and The Aborts List

The GRT stores the txId of the last transaction that read a particular item. When a new reader
is being inserted, the read-number column will be updated with the SON of this previous reader.
Figure 5.1(b) shows that the getTXFromID method is used to obtain the Descriptor of a
past transaction using its txId. However, as mentioned in Section 5.3, it is possible that the
same Descriptor stores the metadata of a new active transaction of the same thread, but not
the metadata of the reader transaction that is being looked for.
Therefore, in order to update the read-number in the GRT, a `getSONFromID` method is used to obtain the SON of a past transaction using its `txId`. For this purpose, an *SON Table* is used to map `txIds` of transactions to their SONs. When a transaction commits, it updates the entry in the SON Table corresponding to its `txId` with its SON. Similar to how shared addresses map to `SharedItem`s, aliasing over the SON Table entries are possible. That is, more than a single transaction id can map to the same entry. Thus, similar to how write-numbers and read-numbers are updated, the SON Table entries are also updated conservatively; an entry is updated only if the current entry is lower than the SON of the committing transaction. This approach again causes some small number of superfluous aborts but it guarantees safe execution. This requires changes in `notifyReaders` method. Figure 5.4 shows the steps of the concurrent `notifyReaders` method.

```plaintext
notifyReaders(tx) {
    for each sharedItem i in tx.writeSet
        for each thread tid
            readerTXID = GRT.activeReaders[i.id][tid]
            readerTX = getTXFromID(readerTXID)
            if readerTX.txId == readerTXID
                updateUpperBound(readerTX, tx.son)
            else if readerTX is COMMITTING
                Abort(tx)
            else if readerTX.txId != readerTXID
                readerSON = getSONFromID(readerTXID)
                if readerSON >= tx.SON
                    Abort(tx)
}
```

Figure 5.4: Concurrent `notifyReaders`.

If a transaction is aborted, it does not update the SON Table. However, other transactions calling `getSONFromID` with the `txIds` of these aborted transactions may receive non-zero SONs due to aliasing over the SON Table, causing unnecessary serializations. In order to prevent this, a small list of aborted `txIds` is used to identify if a transaction is aborted. Thus, `getSONFromID` returns zero if a transaction is found to be aborted.

The described method of identifying the past transactions is an important optimization. Otherwise, GRT must store a pointer to `Descriptor`s rather than storing unique `txIds`, and in order to distinguish transactions from each other, a new `Descriptor` must be created.
every time a new transaction is started. Moreover, determining when it is safe to free these Descriptor objects that belong to past transactions is a complex problem.

5.7.3 Active Readers in the GRT

An important scalability bottleneck in TSTM is the notifyReaders operation performed at commit time as shown in Figure 5.2(b). This operation requires iterating over the active readers column of GRT for each item written to by the committing transaction. In order to lower the latency of the notifyReaders operation, the active readers column in the GRT is organized as a byte-addressable table. That is, GRT stores txIds of reader transactions as a single byte. Thus, when active readers are iterated over during notifyReaders, eight txIds can be read with a single load instruction, thus reducing the number of memory accesses by eight fold.

This optimization significantly reduces the latency of the commit operation which otherwise become a performance bottleneck for scalability. On the other hand, it increases aliasing over the SON Table as txIds can only be one byte long, and hence every 256th transaction has the same txId.

5.7.4 Memory Operations and Atomicity

Memory operations inside transactions can increase the memory print of target applications significantly or even cause errors if they are not handled properly. The main issues are (1) freeing memory blocks allocated inside a transaction if the transaction aborts, (2) reclaiming memory blocks freed inside transactions if the transaction commits, (3) making sure that no transaction is still accessing the memory blocks that are being reclaimed. For this purpose, TSTM provides transactional tx-malloc and tx-free operations. Memory allocations are served from thread private pools. Memory blocks allocated or freed inside transactions are recorded. Allocated blocks are transferred to a reclaim list if the transaction aborts; similarly, freed blocks
are transferred to the reclaim list if the transaction commits. Any memory block stored in the reclaim list is only freed if all the transactions that were active when the memory operation is performed are either committed or aborted. This makes sure that blocks are still not used by other transactions when they are reclaimed. This method of handling transactional memory operations is similar to the method used by RSTM [31].

TSTM only supports weak atomicity [46, 47]. That is, it provides atomicity in the absence of conflicts with non-transactional accesses. Strong atomicity [46, 47] can be provided in the following way. First, TSTM uses lazy VM which eliminates a majority of the atomicity problems, such as speculative lost updates and speculative dirty reads. Second, TSTM can adopt a value-based conflict detection method [36] which validates the read-set using values of shared items. If the value a previously read shared item has changed, this signifies a conflicting write from outside transactions. Finally, techniques proposed for supporting strong atomicity in other STM systems (such as the ones proposed by Shpeisman et al. [46]) can be adopted by TSTM. This is possible because non-transactional accesses do not need to be reordered using SONs. A transaction that conflicts with non-transactional accesses can simply be aborted similar to 2PL.

5.8 Adaptive TSTM

The use of the CS to enforce serializability introduces the overheads described in Section 5.6. Its main performance benefit stems from lowering the abort rate of transactions. Thus, it is likely that applications that have long transactions and high contention will offset the overheads of the use of CS since these applications incur high abort rates and thus high costs of restarting transactions. However, in applications with short transactions that have low contention, it may be possible for the overheads of the use of CS to dominate any benefits of using it. Indeed, in these cases, the use of 2PL is more appropriate.

Therefore, we propose a simple approach that uses 2PL when the abort rate is low and
switches to use CS when the rate is high. Our implementation starts with using 2PL. When the abort rate exceeds a certain threshold $T_h$, the implementation switches to using CS. If the abort rate drops below a threshold $T_l$, the implementation switches back to 2PL. We opt to use two different thresholds $T_l < T_h$, in order to prevent thrashing, i.e., frequent switches between two approaches.

The abort rate is calculated using the total number of aborted and the total number of committed transactions. Both of these metrics are measured by taking the start of the execution as reference, but not the time when the last algorithm switch occurred. Although this approach results in simple implementation, it may not give the best performance for benchmarks that contain phases of execution where the abort rates drastically differ.

Figure 5.5 depicts the algorithm for adaptive concurrency control.

```python
abortRate = numAborted / numCommitted
if concurrency == 2PL
    if abortRate > T_h
        concurrency = CS
    else if concurrency == CS
        if abortRate < T_l
            concurrency = 2PL
```

Figure 5.5: The algorithm for adaptive concurrency control.
Chapter 6

Relaxed Concurrency in Hardware

We refer to our hardware TM system that implements the relaxed SON concurrency control algorithm as SONTM.

In this chapter, we describe the hardware additions that are needed to implement the SON algorithm. More specifically, we describe the general requirements imposed by the use of SONs, the hardware components that implement these requirements, the steps taken by each transactional action, optimizations that improve performance, how race conditions are handled, expected performance tradeoffs, and how the SON algorithm can be implemented on top of a directory-based protocol.

6.1 Base Hardware TM Support

We extend a base hardware EL system with eager CD, lazy VM and requester-wins policy\(^1\). We opt to use eager CD because it meshes well with cache coherence protocols on existing multiprocessor systems; conflicts are detected as memory accesses are occurring using cache coherence messages. Similarly, we opt to use lazy VM because it allows the isolation of transactions until commit time when SONs are assigned to serialize the transactions. In contrast,\(^1\)

\(^{1}\)Similar to our software system, nested transactions are not supported and each processor runs a single transaction at a time.
eager VM systems need a mechanism to handle the side effects of non-serializable transactions until they abort at commit time, as mentioned in Section 4.6.

The EL base system is similar to that of Bobba et al. [20] and is shown in Figure 6.1(a). It extends a typical multiprocessor architecture by adding: a read-set and a write-set for keeping track of reads and writes of transactions, a checkpoint register file for checkpointing registers at the start of transactions and a write-log for storing speculative writes of transactions. The architecture also provides support for conflict detection, aborts and atomic commits [20].

Transactionally stored data is kept in a FIFO write-log, as dictated by the lazy VM scheme. Further, the read-set and the write-set are implemented using Bloom filters as opposed to cache line tags. The use of Bloom filters decouples conflict detection from cache states, as described by Yen et al. [22]. In our base system, we assume perfect bloom filters with no false positives, which ignores the impact of conflict detection mechanisms on performance, which is outside the scope of this work. Having false positives on the read-set and the write-set would make the same impact as if the target application had more conflicts. This may increase the benefits of using a more relaxed concurrency control algorithm such as CS.

Conflicts are detected by keeping track of cache coherence messages at the cache line granularity. When a processor receives a get-shared message for an address from a remote processor, it checks its write-set for possible RAW conflicts. If the write-set includes the requested address, a RAW conflict exists and the processor aborts the receiving transaction, as per the requester-wins policy. Similarly, when a processor receives an invalidate or a get-exclusive message for an address, it checks its read-set and its write-set for possible WAW and WAR conflicts. If either the read-set or the write-set include the requested address, the receiving transaction aborts. In order to ensure that processors receive cache coherence messages, even after a speculatively loaded or stored line is evicted from the cache, the states of such lines are upgraded to a sticky state [22] which allows conflict detection.

When a transaction aborts, it flushes its write-log, restores the register checkpoint, restores the program counter, clears its read-set and its write-set, and then restarts. We use an exponen-
tial backoff for transaction restarts.

When a transaction commits, it iterates over its write-set and commits its speculative stores into the memory. It then clears its read-set and its write-set.

In addition, the base system supports the use of commit tokens and the broadcast and NACK messages, both standard techniques used in shared memory processors and are necessary for our implementation of the SON algorithm, as will be described in the next section.

SONTM adopts strong atomicity [46, 47] provided by the EL base system. That is, when a transaction receives an invalidation message (which signifies a conflict) from a non-transactional access, it aborts. Further, when a processor that runs non-transactional instructions receives an invalidation message from a transaction, it raises exception.

### 6.2 Hardware Components

SONTM aims to serialize transactions according to the order of their conflicting accesses using SON lower and upper bounds. Thus, the main challenge is to able to handle every possible schedule for any two conflicting transactions. Two transactions can conflict in 9 different schedules under lazy VM, as illustrated in Figure 4.1. The serializations in schedules 1-3 are handled by updating the lower bound of the active transaction (TX2 in the figure) with the write and read-numbers of addresses accessed by the two conflicting transactions. Hence, SONTM must implement a write-number table and a read-number table to keep the write-numbers and read-numbers for shared addresses.

In contrast, schedules 4-9 involve two conflicting active transactions. The order in which these transactions commit determines how they will be serialized. When one of the conflicting transactions commits, the other transaction’s lower or upper bound must be updated with the SON of the committing transaction. SONTM employs a broadcast mechanism that is used by the committing transaction to send its SON to all other transactions in the system. Each receiving transaction determines whether it has conflicts with the committing transaction and
the type of these conflicts. For this propose, SONTM uses a set of conflict flags for each transaction that keep track of the type and the rival processor for each of its conflicts.

They key components of SONTM are shown in Figure 6.1(b). They consist of SON registers, the read history tables, and the conflict flags in each processor and the virtual write-number table in the virtual memory.

### 6.2.1 SON Registers

Two special purpose registers $lb$ and $ub$ are used to store the SON lower and upper bounds of the current active transaction. The $lb$ register is initialized to 0 and can only increase. The $ub$ register is initialized to the largest possible value (0xffffffff for a 32-bit system) and can only decrease. In addition, the $son$ register stores the SON of the current transaction at commit time.

### 6.2.2 The Virtual Write-Number Table

It is necessary to keep write-numbers for each shared address in order to serialize transactions after committed writers (Schedules 1-2 in Figure 4.1). Since transactions in lazy VM TM systems store each address in their write-sets into the memory at commit time, it is convenient to keep the write-numbers of these addresses also in the memory, which results in simpler hardware design. Therefore, we use a write-number table in the virtual address space at a location known to all transactions and its entries are initialized to 0. Virtual space addresses are hashed into the table using a subset of the address bits.

Write-numbers are stored into the table at commit time. When the committing transaction iterates over the write-log to commit the new values into the memory, it also calculates the address where the write-number is located in the table for each address in the write-log. While the write-log is flushed into the memory, the write-numbers are also updated.

Write-numbers are loaded when a load instruction is issued and during write-set validation. For a specific cache line address being accessed, the address of the associated write-number
Figure 6.1: Hardware support overview.
is calculated and loaded from the memory. This write-number is then used to update the \( lb \) register.

Loading and storing write-numbers in the table cause overheads due to accesses to the memory. These overheads can be minimized by overlapping the write-number accesses with the transactional data accesses. Further, since write-numbers are kept in the memory, they pollute the caches leading to cache misses. We take into account all these factors when we evaluate the performance of the SONTM in Section 7.2.

Due to the use of hashing, it is possible for more than one cache line to map to the same table entry resulting in aliasing. However, it is important to realize that the aliasing does not lead to incorrect execution but only to unnecessary aborts. This is because, similar to how write-numbers are updated in TSTM, the write-numbers in SONTM are also only updated with larger values. Thus, a transaction \( TX \) will always be forward-serialized after a past transaction \( TX_a \) if it accesses an address \( a \) which was committed by \( TX_a \). However, if \( a \) aliases with another address \( b \) in the table, \( TX \) may also serialize after \( TX_b \) that committed \( b \) in the past (if \( TX_b \)’s SON is higher than \( TX_a \)’s SON), causing possibly an unnecessary abort. We evaluate the impact of the size of the table and the address bits used for hashing and our results show that these parameters have little impact on performance.

In addition, it is possible for a race condition over a table entry to exist between a committing transaction and transactions that are trying to read the same entry in the table. The order of accesses to the table determines how these transactions will be serialized. We expect the hardware commit operations to be fast and hence these race conditions to be rare. Thus, we simply opt to abort a transaction when it conflicts with a committing transaction, which is the only scenario under which such race conditions can occur.

### 6.2.3 The Read History Table

The SON algorithm requires read-numbers to be kept for each shared address in order to serialize writer transactions after committed readers (Schedule 3 in Figure 4.1). Similar to how
write-numbers are stored in a write-numbers table, it is desirable to keep a read-numbers table where each memory address is associated with a read-number. However, this approach incurs significant overheads because it requires iterating over the read-set of a transaction at commit time in order to update the read-numbers in the table.

Thus, rather than keeping a global read-numbers table, we opt to distribute the read-numbers over processors, using history tables, as shown in Figure 6.1(b). Each history table entry (i.e., a history) contains two fields for a specific transaction: the read-set and the SON of that transaction. Because it is not possible to keep histories for all the transactions of a processor, we keep histories only for the last \( n \) successful transactions of a processor. To represent the remaining past transactions, a summary history is kept. The read-set of this summary history matches any address and it contains the largest SON of all the transactions whose histories are not kept. We also implement the read-sets in the histories also as perfect bloom filters; L. Yen et al. [22] shows that using simple filters of size 2 KB delivers close to perfect conflict detection. The history table is shown in Figure 6.2.

The read-set of an active transaction and its SON are stored as the most recent entry in the history table when the transaction commits. The remaining histories are shifted; the oldest entry is used to update the SON of the summary entry and then is discarded.

The histories are accessed during the write-set validation of a transaction. For each address

\[\text{Figure 6.2: History table.}\]
in its write-log, the committing transaction sends a *read-number request* to all the processors and waits until it receives all the responses. Each processor that receives such a request looks up the requested address in its histories. If at least a single match is found, the largest SON of the matching histories is sent back. The requester processor updates its lower bound with the read-numbers sent from all the processors (the largest of these distributed read-numbers is the actual read-number of the address.)

Due to the limited number of histories kept (and due to the inaccuracy of non-perfect bloom-filters if they are used), false conflicts are plausible where an SON value larger than the actual read-number of an address is sent back. These false conflicts may cause unnecessary aborts. However, because read-sets are only used for write-set validation at commit time, the impact of these false conflicts is expected to be minimal.

### 6.2.4 Conflict Flags

Even though conflicts are detected when transactional loads and stores are issued (in accordance with the eager CD), they are handled at commit time when transactional stores become visible and hence impact the serializability of the transactions.

For this purpose, each processor keeps track of its conflicts in two-types of flags: *src* and *dst*, which consists of bits that each represents a conflict with a specific processor. The *src* flags represent the conflicts where the processor is the source of the conflict (i.e., performs the conflicting access first). There are three *src* flags: *src-war*, *src-waw* and *src-raw* for each type of conflict respectively, write-after-read, write-after-write, and read-after-write. The *dst* flags represent the conflicts where the processor is the destination of the conflict (i.e., performs the conflicting access last). The destination processor records the conflict by the type of the instruction that caused it. *dst-read* represents conflicts caused by issuing a load instruction, and *dst-write* represents conflicts caused by issuing a store instruction.

Conflicts are detected by keeping track of cache coherence messages. When a processor receives a *get-shared* signal caused by a remote load instruction, it checks its write-set for a
possible conflict. If a conflict is detected, it records the conflict by setting the corresponding bit in its src-raw flag, signifying that this processor was the source processor in read-after-write conflict. It also sends back a NACK signal to the remote processor so that that remote processor can record the conflict in its dst-read flag. Similarly, the read-set (write-set) is checked for possible conflicts when a processor receives an invalidate or a get-exclusive signal caused by a remote store instruction. In this case, the src-war (src-waw) flag of the receiving processor and the dst-write flag of the issuing processor are updated. Table 6.3 summarizes how the flags are set for each type of conflict.

### 6.3 Transactional Actions

In this section, we describe transactional actions for SONTM. The steps taken for each action are shown in Figure 6.3.

**TX-BEGIN.** The lb and ub registers are initialized to 0’s and 0xffffffff’s respectively.

**TX-LOAD(addr).** The transaction issues a load for the write-number of the addr from the virtual write-number table (1 in Figure 6.3(a)). When the write-number is received 2, the transaction updates its lb flag with the write-number (if it is higher) 3 and asserts that lb < ub – 1; otherwise, it aborts. The read-set is also simultaneously updated to include the cache line address of addr 4.

If the transaction receives a NACK signal from a remote processor 5 in response to the
(a) Load.  
(b) Store.  
(c) Validation.  
(d) Commit.

Figure 6.3: SONTM transactional actions.
load instruction it issued, this represents a RAW conflict. Thus, it records the conflict by setting the corresponding bit in its dst-read flag. The remote processor that sent the NACK also records the RAW conflict in its own src-raw flag by setting the appropriate bit for the load-issuing processor. This conflict will be handled (i.e., conflicting transactions will be serialized) when one of the conflicting transactions commit.

The TX_LOAD cannot complete until the write-number request is serviced and the transaction is verified for serializability. Although this presents a potential source for performance bottleneck, the write-number request can be serviced simultaneously with the actual load from the target address. An optimization, which is described in Section 6.5, is also applied to reduce the number of write-number requests. Our experimental evaluation presented in Section 7.2 shows that the impact of the write-number issue requests on performance is minimal.

**TX_STORE(addr, val).** Since we use a lazy VM system, stores have no impact on consistency until the transaction commits. Therefore, no serializability checks are performed when a store instruction is issued. The cache line address of addr is inserted into the write-set (in Figure 6.3(b)), and val is stored into the write-log together with addr.

**TX_COMMIT.** The transaction starts committing by acquiring a commit token. Then, the transaction validates its stores. For this purpose, it iterates over the write-log and it issues a write-number request (in Figure 6.3(c)) and a read summary request for each address in the log. These are issued simultaneously to overlap memory access latencies. When the transaction receives a write-number or a read-number for a certain address, it updates its lb register and asserts that \( lb < ub - 1 \); otherwise, it aborts immediately. The transaction waits in the validation phase until write-numbers and read-numbers for all the addresses in the write-log are received. Then, it starts the real commit phase. First, it selects its SON and broadcasts it to all the other processors (in Figure 6.3(d)). Then, it saves its read summary in the history tables. Next, it iterates over the write-log once again. During this iteration, it overwrites the write-numbers in the virtual table and commit values into the memory. The commit token guarantees that the write-numbers have not changed since the transaction
starts its commit. The transaction releases the commit token when all the write-numbers have been updated. At this point, it also clears its read-set and its write-set, flushes its write-log and clears its conflict flags. If a processor receives an SON-broadcast message from a committing processor, it checks its outstanding conflicts with this processor. Table 6.5 shows what action is taken for each recorded conflict type.

The above mentioned hardware scheme is complete; i.e., it serializes transactions according to their conflicting actions for every conflict type. More specifically, of all the nine possible conflicting schedules of two transactions shown in Figure 4.1, Schedules 4-9 are handled by the active transaction when the other transaction broadcasts its SON. Further, Schedules 1-3 are handled by updating the lower bounds using the write-numbers and read-numbers of accessed memory addresses.

Further, since SONTM builds upon a base HTM system, it is possible to adjust the concurrency algorithm separately at each processor. That is, some transactions can use 2PL and abort when they receive conflicting coherence requests, while others executing on different processors use the SON algorithm. This can be useful in two ways. First, in applications that contain transactions with different characteristics, the user can choose to execute the transactions with low contention with 2PL, disabling SONTM hardware components, and thus avoiding the SONTM overheads and saving power. Second, adaptive techniques can be implemented to decide which concurrency algorithm to use based on the characteristics of the target application.
6.4 Race Conditions

In SONTM, conflicts are detected using cache coherence messages. Therefore, it is important that the SON bounds of transactions are updated in accordance with the order of these cache coherence messages. Race conditions occur because memory operations and SON operations of transactions can overlap. The race conditions are of the following types.

1. **Committing Writer vs. Committing Writer.** This race condition occurs because two transactions that store to the same address can commit simultaneously. It is handled by simply forcing committing transactions to acquire a global token that only allows a single transaction to commit. This approach prevents two transactions from validating and committing the same address for which they executed a store.

2. **Active Reader/Writer vs. Committing Writer.** This race condition occurs because a transaction can access an address while another transaction validates and commits it to the memory. This is handled by simply aborting transactions when they receive NACK messages from transactions that own the global commit token. In other words, accesses conflicting with committing transactions cause transactions to abort.

6.5 Optimizations

A variation of the retiring transactions optimization described in Section 5.7.1 for TSTM can also be used in SONTM. This optimization is based on the fact that a transaction cannot be part of a cycle in the serializability graph in the future (i.e., can retire) if at the time it committed there were no active transactions serialized before it. That is, there are no active transactions with an upper bound lower than the SON of the committed transaction. Thus, we can say that if none of the active transactions running in the system have their upper bounds changed (i.e., their upper bounds are still equal to $\infty$ initialized at start time), this means that all the transactions that committed in past are retired. Consequently, write-numbers and read-numbers
kept up to this point are obsolete; a transaction does not need to update its lower bound neither with the write-number nor with the read-number of the address it is accessing.

This optimization is applied in SONTM with a straightforward implementation. Transactions keep track of the upper bound updates that occur in the system. When a transaction is about to issue a request to load the write-number or to receive the read-number of an address, it checks if there are any active transactions with their upper bounds set. If this is not the case, this means all the past transactions are retired, and hence, the transaction does not issue the request for the write-number or the read-number.

This simple optimization can reduce the overheads of SONTM significantly by reducing the memory accesses caused by write-number loads and by making commits faster (fewer read-numbers are requested). This is because while transactions are running in isolation with no conflicting accesses, transactions are retired, which eliminates the need to access the write-numbers and read-numbers. The write-numbers and read-numbers are only needed when there are conflicting transactions in the system, which would have been aborted by a standard 2PL implementation. Thus, the overheads of accessing write-numbers and read-numbers are only incurred when it is possible to avoid the aborting of conflicting transactions.

### 6.6 Qualitative Performance Analysis

The main performance benefit of the SON algorithm is transparently reducing the abort rates in the target application. This improves the performance of the application because execution cycles spent for aborted transactions are reduced. The performance impact of this depends on the size of the transactions that would otherwise be aborted. Thus, this can be significant in applications with long transactions and that have high contention.

The hardware implementation of the SON algorithm described earlier in this chapter incur some overheads. The most significant of such overheads is the cache misses caused by accesses to the virtual write-number table. For every transactional memory operation, the corresponding
write-number needs to be loaded from the memory. This increases the latencies of the memory operations as well as it increases the cache misses in the application by polluting the caches with write-numbers. Requests for read-numbers at commit time also introduce overheads. However, our experimental evaluation presented in Section 7.2 shows that the optimization described in Section 6.5 is effective in reducing most of the SONTM overheads caused by write-number and read-number requests.

The implementation of the SON algorithm also introduces overheads in the commit operation. These overheads consist of serialization of commit operations using commit tokens, broadcasting SONs at commit time, stalling commit operations until the SON broadcast messages are received by every other processor, and storing SONs into the write-number table.

### 6.7 Implementation in Directory Based Systems

#### 6.7.1 Issues

The hardware scheme described earlier in this chapter works transparently with systems that use a snoopy broadcast-based cache coherence protocol. This is because, in such systems, conflicts can easily be detected by monitoring the cache coherence messages on the bus. All the processors observe the cache coherence requests on the bus, and conflicts are simply detected by checking the read-sets and write-sets for the requests received.

In contrast, in an EL hardware system that uses a directory based protocol, a processor that runs a transaction will not receive coherence messages for a cache line that was invalidated by a remote store instruction. Hence, the processor cannot detect future conflicts for that cache line. In a hardware TM system that implements the 2PL concurrency algorithm, this does not present any challenge since a transaction that receives an invalidate message always aborts. However, in SONTM, a transaction stays active even after its cache lines are invalidated, yet it cannot detect any future conflicts for these invalidated cache lines. Therefore, we need to make sure that the incompleteness of the conflict detection will not cause any serializability problems
without making any changes to the cache coherence protocol or to the directory. For instance, Figure 6.4 depicts the three schedules where a RAW, a WAW, and a WAR conflicts between TX1 and TX3 are not detected because the store access of TX2 invalidates TX1’s cache line.

### 6.7.2 Hardware Support

Detecting conflicts in a directory based system requires additional hardware support. However, this additional hardware support is also minimal. We simply augment, with extra information, the conflict flags structure, the NACK messages, and the commit broadcast messages. The other components of SONTM (such as the virtual write-number table, the read history table and the son registers) are not modified.

In the directory protocol scheme, conflicts are recorded only by the destination transactions (i.e., transactions that perform the conflicting access last). Thus, the src conflict flags are not used. Destination transactions also record the type of the conflicts. For this purpose, the source transaction sends back an appropriate NACK message that signifies the type of the conflict. In addition, with the NACK message, the source transaction also sends back the list of the past accessor of the specific cache line address. That is, it sends back two bit vectors that represent the transactions that previously read or written to the cache line. This information is generated from the conflict flags. As a result, the information of the readers and writers of an address is propagated from transactions that access the address earlier to the transactions that access the address later. Thus, the transaction that owns the cache line has the most complete information.
of the readers and the writers of that cache line address. This provides the necessary support for implementing CS on top of a directory based protocol.

In this updated scheme, serializations are handled by the destination transactions at commit time. More specifically, if the source transaction (i.e., performs the access first) of the conflict commits first, the destination transaction simply checks its conflict flags and updates its SON bounds with the SON broadcasted. If the destination transaction of the conflict commits first, the source transaction must update its SON bounds. However, because the source transaction does not have any records of the conflict, the SON-broadcast message of the destination transaction also includes two bit vectors that inform the receiving processors if they should update their lower or upper bounds with the SON included in the broadcast message.

Below, we discuss in detail how the hardware components of SONTM are augmented to provide the support in directory based systems.

- **Conflict Flags.** There are three dst flags: dst-war, dst-waw and dst-raw for each type of conflict respectively, write-after-read, write-after-read, and read-after-write. Conflicts are detected by keeping track of cache coherence messages. When a processor receives a get-shared signal caused by a remote load instruction, it checks its write-set for a possible conflict. If a conflict is detected, it sends back a NACK-WR signal to the remote processor (the destination of the conflict) so that the remote processor can record the conflict in its dst-raw flag. Similarly, the read-set (write-set) is checked for possible conflicts when a processor receives an invalidate or a get-exclusive signal caused by a remote store instruction. The dst-war (dst-waw) flag of the issuing processor is updated. Table 6.3 summarizes what signals are sent and the conflict flags that are set for each type of conflict.

- **NACK messages.** The NACK messages contain two bit vectors, past-readers and past-writers, which consist of bits that each represents a transaction previously accessed (read or written to respectively) an address. These bit vectors are generated from the dst con-
Table 6.3: Flags that are set for each type of conflict.

<table>
<thead>
<tr>
<th>Received Signal</th>
<th>Condition</th>
<th>Conflict Type</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>get-shared</td>
<td>write-set matches</td>
<td>Read-After-Write</td>
<td>NACK-WR is sent</td>
</tr>
<tr>
<td>invalidate</td>
<td>read-set matches</td>
<td>Write-After-Read</td>
<td>NACK-RD is sent</td>
</tr>
<tr>
<td>get-exclusive</td>
<td>read-set-matches</td>
<td>Write-After-Write</td>
<td>NACK-WR is sent</td>
</tr>
<tr>
<td>NACK-WR</td>
<td>write-set-matches</td>
<td>Read-After-Write</td>
<td>NACK-WR is sent</td>
</tr>
<tr>
<td>NACK-RD</td>
<td>issued a load</td>
<td>Write-After-Read</td>
<td>dst-raw is set</td>
</tr>
<tr>
<td>NACK-WR</td>
<td>issued a store</td>
<td>Write-After-Read</td>
<td>dst-war is set</td>
</tr>
</tbody>
</table>

Conflict flags. For instance, if the dst-war flag is set for a remote transaction, this means that this remote transaction has read this particular address in the past. Similarly, if the dst-waw or dst-raw flags are set for a remote transaction, this means this remote transaction has written to this particular address in the past. The extra past-readers and past-writers information carried increases the size of NACK messages by only 4 bytes, thus do not cause bus contention. When a transaction receives the NACK message, it updates its dst flags according to (1) the type of the NACK message (i.e., NACK-ST or NACK-LD), and to (2) the past-readers and the past-writers bit vectors contained in the NACK message. As a result, the transaction has a complete record of the conflicts for this particular address.

- **Commit Messages.** At commit time, a transaction determines which transactions should update their SON bounds depending on the types of the conflicts. It includes this information in the broadcast message with two bit vectors called uptLower and uptUpper. Table 6.4 shows how these bit vectors are generated using simple bit operations. If a processor receives a SON-broadcast message from a committing processor, it checks its outstanding conflicts with this processor. Table 6.5 shows what action is taken for each recorded conflict type.
Table 6.4: How $uptLower$ and $uptUpper$ bit vectors are set at commit time.

<table>
<thead>
<tr>
<th>Condition</th>
<th>Updated Bit Vector</th>
<th>Schedule (Figure 4.1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$i$th bit in $dst$-$raw$ is set</td>
<td>$i$th bit in $uptLower$ is set</td>
<td>Schedule 4</td>
</tr>
<tr>
<td>$i$th bit in $dst$-$waw$ is set</td>
<td>$i$th bit in $uptLower$ is set</td>
<td>Schedule 5</td>
</tr>
<tr>
<td>$i$th bit in $dst$-$war$ is set</td>
<td>$i$th bit in $uptUpper$ is set</td>
<td>Schedule 6</td>
</tr>
</tbody>
</table>

Table 6.5: Actions taken by processor $i$ when it receives a SON broadcast message from processor $k$.

<table>
<thead>
<tr>
<th>Matching Conflict Flag</th>
<th>Updated Register</th>
<th>Schedule (Figure 4.1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$i$th bit is set in $uptLower$</td>
<td>$lb$</td>
<td>Schedule 4,5</td>
</tr>
<tr>
<td>$i$th bit is set in $uptUpper$</td>
<td>$ub$</td>
<td>Schedule 6</td>
</tr>
<tr>
<td>$k$th bit is set in $dst$-$raw$</td>
<td>$ub$</td>
<td>Schedule 7</td>
</tr>
<tr>
<td>$k$th bit is set in $dst$-$waw$</td>
<td>$lb$</td>
<td>Schedule 8</td>
</tr>
<tr>
<td>$k$th bit is set in $dst$-$war$</td>
<td>$lb$</td>
<td>Schedule 9</td>
</tr>
</tbody>
</table>
Chapter 7

Experimental Evaluation

This chapter presents an experimental evaluation of the software and the hardware implementations of the SON algorithm, which were described in Chapter 5 and in Chapter 6 respectively. More specifically, Section 7.1 presents the experimental evaluation for the software implementation and Section 7.2 presents the experimental evaluation for the hardware implementation.

7.1 Evaluation of TSTM

In this section, we provide an experimental evaluation of TSTM and thus the potential benefits of using CS in an STM system to enforce serializability. We measure the performance of standard benchmark applications. We compare the performance of these benchmarks using TSTM against their performances using TL2, a state-of-the-art blocking STM system. The results show that the overheads of checking for CS in TSTM do increase the execution time of transactions. However, the benefits of reducing the abort rates outweighs these overheads, resulting in improved performance for benchmarks that have high abort rates and long transactions. For the benchmarks that have low abort rates or short transactions, TL2 outperforms TSTM due to its highly optimized lock-based implementation.

The remainder of this section is organized as follows. Section 7.1.1 describes the evaluated
STM systems. Section 7.1.3 presents the experimental platform. Section 7.1.4 defines the metrics used in our evaluation. Section 7.1.5 lists the benchmarks used and describes their characteristics. Finally, Section 7.1.6 presents and discusses the experimental results.

### 7.1.1 Evaluated STM Systems

We evaluate several implementations of TSTM and other systems. These implementations are described below.

- **base** is a version of TSTM that uses 2PL instead of CS to enforce serializability. More specifically, when a WAW conflict is detected the transaction that performed the access later is aborted. Similarly, when a WAR conflict is detected, the reader transaction is aborted rather than updating its upper bound with the SON of the writer transaction. Otherwise, **base** is identical to TSTM in every other aspect (such as data structures, transactional actions, etc.). Because this version uses the same data structures, it incurs similar cache affects as TSTM. However, it does not incur the overheads of updating SON lower and upper bounds, write-numbers and read-numbers. Thus, we use this version to measure the impact of testing for CS and the impact of reducing abort rates on the performance.

- **cs** is the version of TSTM system described Chapter 5. It implements CS using the SON technique and uses visible reads.

- **adaptive** is the version of TSTM system that uses the adaptive approach described in Section 5.8.

- **tl2-lazy** is TL2, a state-of-the-art blocking (lock-based) STM system [12] commonly used for evaluating novel TM systems in the literature [33, 39]. It implements 2PL concurrency model, uses word-based granularity and automatically acquires locks to protect writes to shared variables. TL2 uses versions associated with each shared variable
to keep track of the writes to shared variables (i.e., if the version of a shared variable is changed, this means the shared variable has been written to by another transaction since the read action). A global version is used to keep track of transactions commits (i.e., if the global version is not changed, this means that no transaction has committed since the last version check, and hence, the read-set is guaranteed to be consistent). However, the use of global version is only an optimization that eliminates the need for read-set validation, but not a relaxation of the concurrency model. That is, transactions still abort every time a conflict is detected in accordance with the 2PL algorithm. This optimization is further discussed in related work (see Chapter 8).

We use an official release of TL2 [11] which is used to evaluate the STAMP benchmarks [33]. This version is compiled using the lazy option without any changes to the source code. That is, it uses lazy acquire for writes, i.e., writes are buffered during transaction execution. At commit time, locks associated with each shared variable written by the committing transaction are acquired. The read-set is also validated at commit time by making sure that any shared variable read by the transaction has not changed since the read action. Read actions require searching through the write-set for finding whether the shared variable currently being read was written earlier by the same transaction. This write-set search and the read-set validation constitute the main overheads of this lazy TL2 implementation.

- **tl2-eager** is a version of TL2 compiled using the eager option without any changes to the source code. It uses eager acquire for writes, i.e., locks associated with shared variables are acquired during transaction execution, rather than during commit. Thus, a write-set search is not required for read actions as transactions that write to shared variables can easily be identified by verifying the owner of the associated locks.
Table 7.1: The implementation parameters of TSTM.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size of the GRT</td>
<td>256K</td>
</tr>
<tr>
<td>Number of SharedItems pre-allocated</td>
<td>256K</td>
</tr>
<tr>
<td>Hashing of shared addresses to SharedItems</td>
<td>index = (addr &gt;&gt;&gt; 4)</td>
</tr>
<tr>
<td>Size of the SON Table</td>
<td>256K</td>
</tr>
<tr>
<td>Types of write-number, read-number and SON bounds</td>
<td>4-byte long unsigned integers</td>
</tr>
<tr>
<td>Low threshold for adaptive</td>
<td>( T_l = num_threads \times 0.0025 - 0.01 )</td>
</tr>
<tr>
<td>High threshold for adaptive</td>
<td>( T_h = num_threads \times 0.005 )</td>
</tr>
</tbody>
</table>

### 7.1.2 TSTM Parameters

We observe that the choice of parameters has only a small impact on the performance. Nevertheless, since the data structures are kept in virtual memory, we choose reasonably large sizes in order to reduce the aliasing. The implementation parameters of TSTM are listed in Table 7.1.

We also observe that there is no linear correlation between the sizes of the data structures and the performance. In fact, using a smaller data structure (such as GRT) may cause more transactions to abort unnecessarily by increasing aliasing, however, it may also increase performance by reducing the cache misses.

The threshold values of the adaptive implementation are chosen by comparing the performances of base and cs versions. It is desirable to use a formula that correlates the abort rates with the best performing algorithm. The experimental results show that it possible to find such a formula that works for a wide range of benchmarks.

### 7.1.3 STM Experimental Platform

The experiments are conducted on a real Sun SPARC Enterprise T5140 Server. The specifications of this platform are shown in Table 7.2. We apply thread binding to assign threads to specific processors so that each thread executes on a separate core. However, since our platform only contains 2x8=16 cores in total, for the experiments that run with 32 threads, each
core is assigned with two threads.

7.1.4 STM Evaluation Metrics

We evaluate the characteristics of the benchmarks and their performances with STM systems using several performance metrics. These metrics are:

- **The total execution time** is the time measured from the initialization of the STM libraries until the shutdown of the libraries. Thus, it excludes certain initialization operations such as opening and reading from the input files.

- **The abort rate** is calculated as the ratio of the number of aborted transactions to the total number of transactions successfully committed. Thus, an abort rate of 1 signifies that each transaction aborts once on average before it successfully commits.

- **The speedup** is calculated as the ratio of the total execution time of the benchmark to the total execution time of the same benchmark measured using the base STM system on a single thread.

- **The slowdown** is calculated as the ratio of the total execution time of the benchmark executed using the base STM system on a single thread to the total execution time of the sequential version of the same benchmark which does not include any synchronization primitives (i.e., neither locks nor transactions).

7.1.5 STM Benchmarks

We assess the impact of the SON algorithm on performance using the set of benchmarks described below. The benchmarks consist of a synthetic microbenchmark (list) and the entire STAMP benchmark suite [33]. We elect these benchmarks because they contain both short-running transactions with low contention as well as long-running transactions with high contention, thus, collectively the benchmarks demonstrate the advantages and disadvantages of
Table 7.2: Specifications of the experimental platform for evaluating STM systems.

<table>
<thead>
<tr>
<th>Component</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Server Name</td>
<td>Sun SPARC Enterprise T5140 Server</td>
</tr>
<tr>
<td>Processors</td>
<td>two 1.2 Ghz UltraSparc T2+ chips (Victoria Falls), each with 8 cores</td>
</tr>
<tr>
<td>Cores</td>
<td>capable of Chip Multi Threading with 8 hardware threads</td>
</tr>
<tr>
<td>L1 Cache</td>
<td>16KB per core on-chip</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>Shared 4 MB on-chip 16-way with 64B cache lines</td>
</tr>
<tr>
<td>Main Memory</td>
<td>32 GB of memory (2GB per core)</td>
</tr>
<tr>
<td>Memory Bandwidth</td>
<td>32 GB/sec</td>
</tr>
<tr>
<td>Virtual Memory</td>
<td>64 KB pages</td>
</tr>
</tbody>
</table>

using CS for concurrency control. The following describes these benchmarks and some of their characteristics.

- **list** implements an ordered linked-list, where each node contains an unsigned integer key and the nodes are singly-linked to each other. This benchmark exhibits long transactions and high abort rates, hence, is a good candidate for the use of CS.

- **bayes** implements an algorithm for learning the structure of Bayesian networks from observed data. The Bayesian network is represented as a directed acyclic graph, with a node for each variable and an edge for each conditional dependence between variables. A transaction is used to protect the calculation and the addition of a new dependency. The calculation of new dependencies takes up most of the execution time. Thus, **bayes** spends almost all its execution time in long transactions. This benchmark also exhibits high amount of contention. Thus, it is a good candidate for the use of CS.

- **vacation** implements an online transaction processing system by emulating a travel reservation system. The system is implemented as a set of trees that keep track of customers and their reservations for various travel items (such as cars, accommodation, etc.). Each client request is enclosed in a coarse-grain transaction to ensure validity of the database. This application spends considerable amount of time in transactions. The transactions are of moderate length and exhibit high contention. Thus, this application
can benefit from the use CS.

- **labyrinth** implements a maze path-solving algorithm. The main data structure is a three-dimensional grid that represents the maze. Each thread grabs a start and end point that it must connect by a path of adjacent maze grid points. The calculation of the path and its addition to the global maze grid are enclosed in a single transaction. Most of the execution time is spent for the path calculation. Thus, the application has very long transactions and the amount of contention is very high. This makes this it a good candidate for the use of CS.

- **yada** implements an algorithm for Delaunay mesh refinement [43]. Transactions enclose accesses to the work queue, as well as refinement of triangles. The application spends almost all its execution refining triangles, which makes this application have long transactions. The application also exhibits high contention.

- **intruder** emulates a signature-based network intrusion detection system that scan network packets against a known set of intrusion signatures. The *capture* and *reassembly* phases of packet processing are included inside transactions. This application has short transactions and may incur moderate to high levels of contention.

- **genome** matches a large number of DNA segments to reconstruct the original source genome. Transactions are used to protect accesses to the set of unique gene segments as well as to the global pool of unmatched gene segments. This benchmark has transactions of moderate length with little contention.

- **kmeans** groups objects in an N-dimensional space into K clusters. Each thread processes a partition of the objects iteratively. A transaction is used to protect the update of the cluster center that is calculated during each iteration. Little of the total execution time is spent inside transactions. The transactions are very short and the amount of contention is low. Thus, this benchmark is not likely to benefit from the use CS.
- ssca2 constructs an efficient graph data structure using adjacency arrays and auxiliary arrays. Threads add nodes into the graph in parallel and transactions are used to protect accesses to the adjacency arrays. Little time is spent in transactions which are also small, and the amount of contention is low. Thus, similar to \textit{kmeans}, we expect this benchmark not to benefit from the use of CS.

\textit{labyrinth} employs a manual optimization, which privatizes the global grid that represents the maze to be solved before starting each transaction. Hence, transactions only read and write to the global grid for the paths that they route. This greatly reduces the abort rates by preventing transactions from conflicting over the grid when they search all the possible paths for the most optimum one. This privatization optimization requires significant knowledge of the application behaviour and can only be applied by a programmer who understands TM working mechanisms. Thus, in order to evaluate the performance of TSTM for a naively ported version of \textit{labyrinth} that does not employ any optimizations, we also generated \textit{labyrinth-nop} version of this benchmark.

Table 7.3 depicts the parameters given to each benchmark. We choose parameters that were used in the literature [39] or parameters that generate run times long enough (more than 1 second on one thread) to alleviate the side effects of the OS and the memory. However, for \textit{bayes}, the parameters that generate such long running times cause the system to run out of memory during the initial graph generation stage before any transactions are executed. Thus, we exclude this benchmark from our evaluation.

Table 7.4 shows the following metrics for each benchmark run using the \texttt{base} STM with a single thread; the total execution time of the benchmark (Total Time), the ratio of transactions to the total execution time (Xact Ratio), the total number of committed transactions (Xact Count), the average time spend for each transaction (Xact Time), the average number of read actions per transaction (Xact Reads), the average number of write actions per transaction (Xact Writes) and the slowdown.
Table 7.3: Benchmark input parameters for STM systems.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>list</td>
<td>-value-range 0:8192 -insert-delete-lookup-ratio 1:1:1</td>
</tr>
<tr>
<td>vacation</td>
<td>-t 20000 -n 110 -u 30</td>
</tr>
<tr>
<td>labyrinth</td>
<td>-i random-x256-y256-z3-n256.txt</td>
</tr>
<tr>
<td>labyrinth-nop</td>
<td>-i random-x256-y256-z3-n256.txt</td>
</tr>
<tr>
<td>yada</td>
<td>-a 10 -i timeout100000.2</td>
</tr>
<tr>
<td>intruder</td>
<td>-a 10 -l 16 -n 16384 -s 1</td>
</tr>
<tr>
<td>genome</td>
<td>-g 65536 -s 64 -n 1677216</td>
</tr>
<tr>
<td>kmeans</td>
<td>-m 15 -n 15 -t 0.05 -i random-n65536-d32-c16.txt</td>
</tr>
<tr>
<td>ssca2</td>
<td>-s 17 -i 1.0 -u 1.0 -l 3 -p 3</td>
</tr>
</tbody>
</table>

Table 7.4: Benchmark characteristics measured with the base STM system.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Total Time</th>
<th>Xact Ratio</th>
<th>Xact Count</th>
<th>Xact Time</th>
<th>Xact Reads</th>
<th>Xact Writes</th>
<th>Slow-down</th>
</tr>
</thead>
<tbody>
<tr>
<td>list</td>
<td>7.40 s.</td>
<td>99%</td>
<td>100,000</td>
<td>622 μs.</td>
<td>2390.5</td>
<td>0.5</td>
<td>7.40</td>
</tr>
<tr>
<td>vacation</td>
<td>6.59 s.</td>
<td>97%</td>
<td>20,000</td>
<td>667 μs.</td>
<td>2196.2</td>
<td>80.6</td>
<td>6.59</td>
</tr>
<tr>
<td>labyrinth</td>
<td>25.75 s.</td>
<td>99%</td>
<td>514</td>
<td>49972 μs.</td>
<td>91.7</td>
<td>88.2</td>
<td>1.00</td>
</tr>
<tr>
<td>labyrinth-nop</td>
<td>52.37 s.</td>
<td>99%</td>
<td>514</td>
<td>108500 μs.</td>
<td>2293070.0</td>
<td>89.2</td>
<td>1.62</td>
</tr>
<tr>
<td>yada</td>
<td>8.90 s.</td>
<td>95%</td>
<td>105,782</td>
<td>60 μs.</td>
<td>62.5</td>
<td>17.4</td>
<td>2.29</td>
</tr>
<tr>
<td>intruder</td>
<td>1.85 s.</td>
<td>67%</td>
<td>221,359</td>
<td>8 μs.</td>
<td>20.9</td>
<td>2.3</td>
<td>3.43</td>
</tr>
<tr>
<td>genome</td>
<td>382.88 s.</td>
<td>96%</td>
<td>4,516,037</td>
<td>81 μs.</td>
<td>162.6</td>
<td>2.0</td>
<td>2.66</td>
</tr>
<tr>
<td>kmeans</td>
<td>11.48 s.</td>
<td>77%</td>
<td>262,146</td>
<td>38 μs.</td>
<td>25.0</td>
<td>25.0</td>
<td>5.49</td>
</tr>
<tr>
<td>ssca2</td>
<td>11.98 s.</td>
<td>29%</td>
<td>1,549,861</td>
<td>3 μs.</td>
<td>1.1</td>
<td>2.0</td>
<td>1.73</td>
</tr>
</tbody>
</table>

Figure 7.1 depicts the abort rate and Xact Time distribution of each benchmark. The abort rate is measured using the base system at 16 threads, whereas Xact Time is measured using the base system at a single thread. We measure the two metrics at different threads because it is not possible to accurately measure Xact Time at 16 threads without affecting performance.

From the figure, it can be seen that labyrinth-nop has very long transactions and high abort rate compared to the remaining benchmarks. Thus, this benchmark is a good target application for TSTM. Further, list and vacation have high abort rates and long transactions. Thus, these applications are also good candidates to benefit from TSTM. The labyrinth benchmark has long transactions but low abort rates. genome, yada and kmeans have moderately long transactions, but low abort rates, and hence, they may benefit from TSTM. Finally,
intruder and ssca2 have short transactions and low abort rates. Thus, these two applications are not expected to benefit from TSTM.

![Figure 7.1: Abort rate and Xact Time distribution of the benchmarks measured with the base STM system.](image)

7.1.6 Experimental Results

We measure the speedups and the abort rates of each benchmark on 1, 2, 4, 8, 16 and 32 threads, which are shown in Figures 7.2-7.4. Table 7.5 shows the experimental results for all benchmarks and all the STM systems on 16 threads. A number of comparisons can be made from the experimental results.

Comparative Performance of the STM Systems

Below, we compare the performance of the STM systems.

- base vs. cs. The use of CS reduces abort rates in most benchmarks, i.e., in list, vacation, labyrinth-nop, yada, genome, kmeans. The difference is significant in applications with high abort rates such as list, vacation and labyrinth-nop.
The abort rates of these applications are reduced from 1.65 to 0.35, from 1.34 to 0.82, and from 4.49 to 3.29 respectively on 16 threads. The difference in abort rates is small in applications with low abort rates such as yada, genome and kmeans. The abort rates of labyrinth, intruder and ssca2 are not affected by the use of CS because a manual inspection of the conflicts reveals that transactions in these benchmarks are not conflict-serializable when they conflict.

The reduction in abort rates positively impacts the speedup performance. This impact is significant in applications with long transactions such as list, labyrinth-nop, vacation. The performance of these benchmarks are improved by 2.14x, 2.99x and 1.25x respectively on 16 threads. The performance of genome also improves by 1.31x on 16 threads with the use of CS as this application contains transactions of different sizes and the reduction in the abort rates of longer transactions positively impact the performance. The speedups of yada and kmeans, on the other hand, are not affected by the small reduction in abort rates as these applications only contain short transactions.

- **base vs. t12.** The differences in the design choices made to implement our base STM system and t12 lead to different performance characteristics. Mainly, base (as well as cs and adaptive) is an obstruction-free STM system with visible reads and lazy acquire. In contrast, t12 implementations, on the other hand, are blocking STM systems with invisible reads. This causes t12 implementations to perform expensive read-set validation, as described in Section 7.1.1. Further, t12-eager does not require write-set search that is required by our STM implementations as well as by t12-lazy due to the use of lazy acquire.

Thus, base outperforms t12 implementations in labyrinth-nop, vacation and list benchmarks that execute transactions with large number of read actions that lead to expensive read-set validations. The base implementation outperforms t12-eager also for yada which suffer from high abort rates due to eager-acquire. The t12 pro-
duces performance that is close to base for labyrinth and ssca2 benchmarks. The abort rates of both implementations are also close to one another for these benchmarks. However, due to faster transactional actions provided by the invisible reads technique, the tl2 implementations outperform base for kmeans and intruder.

- **adaptive vs. cs.** The adaptive techniques employed by TSTM are effective in matching the best performance of base and cs using the same threshold values for all the benchmarks. While the default 2PL concurrency algorithm is used for ssca2 which has a very low abort rate (below the $T_i$ threshold), the concurrency algorithm used transitions to the SON algorithm which implements CS for the remaining benchmarks applications for which CS produces the superior performance, thus dynamically adapting the concurrency algorithm to the application behaviour.

### Comparative Performance of the Benchmarks

Below, we discuss the performance of each benchmark application separately.

- **list.** This benchmark exhibits high abort rates and contains long transactions. Thus, reduction in abort rates lead to improved performance when the benchmark is run with cs. In fact, the remaining STM systems suffer from high abort rates and their performances do not scale with increasing number of threads.

- **vacation.** This benchmark contains long transactions that abort frequently. tl2-lazy produces very low performance as it suffers significantly from write-set searches performed for large number of read actions in every transaction. tl2-lazy also suffers from very high abort rates (217.93 on 16 threads). tl2-eager outperforms tl2-lazy, however its performance is inferior to the performances of base and cs due its higher abort rates (21.32 on 16 threads) and read-set validations. cs outperforms base solely due to the reduction in abort rates. For instance, the abort rate of base is 1.34 on 16 threads, whereas the abort rate of cs is 0.83 for the same number of threads.
The performances of base, cs and adaptive drops with 32 threads due to the lack of available free cores.

- **labyrinth.** This benchmark contains long transactions with moderate abort rates. Due to the privatization optimization employed in this benchmark, the transactions are not conflict-serializable when they conflict. Thus, the abort rates and the speedup performances do not vary among STM implementations.

- **labyrinth-nop.** This benchmark contains very long transactions with high abort rates. tl2-lazy suffers excessively from write-set searches and, hence, the benchmark does not complete in time comparable to the other STM implementations, hence it is excluded from the evaluation. tl2-eager does not suffer from write-set searches, however, it suffers from excessive abort rates (162.66 on 16 threads). base shows improving performance with increasing number of threads, but it also suffers from high abort rates which prevent its speedup performance from scaling. cs scales up to 16 threads due to its lower abort rates. For instance, the abort rate is reduced from 4.49 to 3.30 with cs. This difference of 1.19 is significant because it means that each transaction restarts on average more than one extra time with the base system.

- **yada.** This benchmark has low abort rates and moderately long transactions. Due to the read-set validations, this benchmarks performs poorly with tl2 implementations. On the other hand, base, cs and adaptive that use visible reads (hence do not require read-set validation) perform well for this application. Because abort rates are not reduced significantly, base and cs present similar speedup performances.

- **intruder.** This benchmark has short transactions and low abort rates. Although tl2-lazy and tl2-eager suffer from higher abort rates compared to base and cs, the differences are not significant. Thus, tl2-eager scales well up to 8 threads and outperforms the other STM systems. tl2-lazy is outperformed by base and cs because
it performs inefficient write-set search. Because the abort rates are not affected by the concurrency algorithm used, base and cs exhibit similar speedup performances. Finally, the throughput of all STM systems drops after 16 threads. This is likely to be caused by the cache effects.

- genome. This benchmark has moderately long transactions and low abort rates. tl2-eager and tl2-lazy caused assertion errors and led to crashes in every experiment we ran. Thus, we exclude the results for these implementations. The abort rates are reduced by the cs compared to base, which leads to improvement in speedups.

- kmeans. This benchmark has moderately long transactions and low abort rates. tl2-lazy produced incorrect output for this benchmark and, hence, its experimental results are excluded. tl2-eager, on the other hand, scales very well with increasing number of threads and outperforms the remaining STM implementations. The abort rates are not affected by the use of CS, thus, the performances of base and cs are close to each other.

- ssca2. This benchmark has short transactions and very low abort rates. All the STM systems scale well with increasing number of threads. tl2-eager and tl2-lazy slightly outperform base and cs due to the invisible reads technique they implement. The abort rates of base and cs are slightly higher than the abort rates of tl2 implementation due to the visible reads. However, the difference is insignificant.
Figure 7.2: Speedup and abort rates for list, vacation, and labyrinth.
Figure 7.3: Speedup and abort rates for labyrinth-nop, yada, and intruder.
Figure 7.4: Speedup and abort rates for genome, kmeans, and scca2.
Table 7.5: Experimental results on 16 threads.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>STM</th>
<th>Commit</th>
<th>Aborts</th>
<th>Time</th>
<th>Abort Rate</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>list</td>
<td>base</td>
<td>100,000</td>
<td>165,801</td>
<td>11.42</td>
<td>1.66</td>
<td>3.45</td>
</tr>
<tr>
<td></td>
<td>cs</td>
<td>100,000</td>
<td>35,294</td>
<td>5.33</td>
<td>0.35</td>
<td>7.38</td>
</tr>
<tr>
<td></td>
<td>adaptive</td>
<td>100,000</td>
<td>38,316</td>
<td>5.31</td>
<td>0.38</td>
<td>7.41</td>
</tr>
<tr>
<td></td>
<td>tl2-lazy</td>
<td>100,000</td>
<td>389,181</td>
<td>20.38</td>
<td>3.89</td>
<td>1.93</td>
</tr>
<tr>
<td></td>
<td>tl2-eager</td>
<td>100,000</td>
<td>313,912</td>
<td>10.83</td>
<td>3.14</td>
<td>3.64</td>
</tr>
<tr>
<td>vacation</td>
<td>base</td>
<td>20,000</td>
<td>26896</td>
<td>2.50</td>
<td>1.34</td>
<td>4.43</td>
</tr>
<tr>
<td></td>
<td>cs</td>
<td>20,000</td>
<td>16,562</td>
<td>1.99</td>
<td>0.83</td>
<td>5.55</td>
</tr>
<tr>
<td></td>
<td>adaptive</td>
<td>20,000</td>
<td>17,991</td>
<td>2.04</td>
<td>0.90</td>
<td>5.43</td>
</tr>
<tr>
<td></td>
<td>tl2-lazy</td>
<td>20,000</td>
<td>4,358,664</td>
<td>202.47</td>
<td>217.93</td>
<td>0.05</td>
</tr>
<tr>
<td></td>
<td>tl2-eager</td>
<td>20,000</td>
<td>426,305</td>
<td>6.05</td>
<td>21.32</td>
<td>1.83</td>
</tr>
<tr>
<td>labyrinth</td>
<td>base</td>
<td>544</td>
<td>193</td>
<td>3.80</td>
<td>0.35</td>
<td>6.77</td>
</tr>
<tr>
<td></td>
<td>cs</td>
<td>544</td>
<td>188</td>
<td>3.68</td>
<td>0.35</td>
<td>7.00</td>
</tr>
<tr>
<td></td>
<td>adaptive</td>
<td>544</td>
<td>196</td>
<td>3.72</td>
<td>0.36</td>
<td>6.91</td>
</tr>
<tr>
<td></td>
<td>tl2-lazy</td>
<td>544</td>
<td>215</td>
<td>3.95</td>
<td>0.40</td>
<td>6.53</td>
</tr>
<tr>
<td></td>
<td>tl2-eager</td>
<td>544</td>
<td>246</td>
<td>4.18</td>
<td>0.45</td>
<td>6.17</td>
</tr>
<tr>
<td>labyrinth-nop</td>
<td>base</td>
<td>22,1374</td>
<td>24,610</td>
<td>1.41</td>
<td>0.22</td>
<td>6.31</td>
</tr>
<tr>
<td></td>
<td>cs</td>
<td>22,1374</td>
<td>21,610</td>
<td>1.45</td>
<td>0.19</td>
<td>6.14</td>
</tr>
<tr>
<td></td>
<td>adaptive</td>
<td>22,1374</td>
<td>24,967</td>
<td>1.43</td>
<td>0.22</td>
<td>6.22</td>
</tr>
<tr>
<td></td>
<td>tl2-lazy</td>
<td>22,1374</td>
<td>119,723</td>
<td>26.71</td>
<td>1.08</td>
<td>0.33</td>
</tr>
<tr>
<td></td>
<td>tl2-eager</td>
<td>22,1374</td>
<td>611,659</td>
<td>24.17</td>
<td>5.55</td>
<td>0.37</td>
</tr>
<tr>
<td>yada</td>
<td>base</td>
<td>4,516,045</td>
<td>363,639</td>
<td>133.45</td>
<td>0.08</td>
<td>2.87</td>
</tr>
<tr>
<td></td>
<td>cs</td>
<td>4,516,045</td>
<td>191,847</td>
<td>101.50</td>
<td>0.04</td>
<td>3.77</td>
</tr>
<tr>
<td></td>
<td>adaptive</td>
<td>4,516,045</td>
<td>199,971</td>
<td>104.68</td>
<td>0.04</td>
<td>3.66</td>
</tr>
<tr>
<td></td>
<td>tl2-lazy</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>tl2-eager</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>intruder</td>
<td>base</td>
<td>262,191</td>
<td>114,876</td>
<td>2.70</td>
<td>0.44</td>
<td>4.26</td>
</tr>
<tr>
<td></td>
<td>cs</td>
<td>262,191</td>
<td>99,192</td>
<td>2.60</td>
<td>0.38</td>
<td>4.41</td>
</tr>
<tr>
<td></td>
<td>adaptive</td>
<td>262,191</td>
<td>98,327</td>
<td>2.59</td>
<td>0.38</td>
<td>4.43</td>
</tr>
<tr>
<td></td>
<td>tl2-lazy</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>tl2-eager</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>genome</td>
<td>base</td>
<td>1,542,528</td>
<td>536</td>
<td>2.83</td>
<td>0.00</td>
<td>4.24</td>
</tr>
<tr>
<td></td>
<td>cs</td>
<td>1,542,528</td>
<td>444</td>
<td>2.95</td>
<td>0.00</td>
<td>4.07</td>
</tr>
<tr>
<td></td>
<td>adaptive</td>
<td>1,542,528</td>
<td>540</td>
<td>2.91</td>
<td>0.00</td>
<td>4.12</td>
</tr>
<tr>
<td></td>
<td>tl2-lazy</td>
<td>1,542,528</td>
<td>147</td>
<td>2.63</td>
<td>0.00</td>
<td>4.55</td>
</tr>
<tr>
<td></td>
<td>tl2-eager</td>
<td>1,542,528</td>
<td>133</td>
<td>2.55</td>
<td>0.00</td>
<td>4.69</td>
</tr>
</tbody>
</table>
Breakdown of Execution

We also evaluate the impact of aborts on performance. For this purpose, we measure, using the `gethrt ime` system call, the total time spent for committed and aborted transactions in list, vacation, and labyrinth-nop benchmarks. Figure 7.5 shows this breakdown of the total execution time on 1, 2, 4, 8, 16, and 32 threads.

On one thread, the total time it takes to execute the transactions is the same for both cs and base systems because no time is spent on aborted transactions. As the number of threads is increased, the ratio of the time spent for aborted transactions increases for both systems. However, this increase is more dramatic for base because its abort rate is significantly higher, as discussed earlier in this section. In fact, on 32 threads, 72% of the 10.162 seconds total execution time is spent by aborted transactions with base in list as opposed to 49% of the 4.848 seconds of the total execution time with cs.
(a) The breakdown of execution for \textit{list}.

(b) The breakdown of execution for \textit{vacation}.

(c) The breakdown of execution for \textit{labyrinth-nop}.

Figure 7.5: The breakdown of execution for \texttt{base} and \texttt{cs} systems.


7.2 Evaluation of SONTM

In this section, we provide an experimental evaluation for SONTM and evaluate the benefits of CS in hardware TM systems. Section 7.2.1 describes the evaluated HTM systems. Section 7.2.2 presents the experimental platform. Section 7.2.3 defines the metrics used in our evaluation. Section 7.2.4 lists the benchmarks used and describes their characteristics. Finally, Section 7.2.5 presents and discusses the experimental results.

7.2.1 Evaluated HTM Systems

We evaluate the following implementations of HTM systems.

- **2pl-EL** is the standard EL hardware transactional system used as the base system to implement SONTM. It is based on the 2PL concurrency algorithm, where each conflict causes an abort. Conflicts are detected using coherence messages. Each processor keeps bloom filters to represent the read-set and the write-set of active transactions. Because false conflicts are outside the scope of this thesis, we use perfect bloom filters that do not cause any false positives. If a transactional cache line is evicted from the cache, its state is changed to a sticky state [22].

- **2pl-LL** is an LL configuration of the hardware TM system used by Bobba et al. [20]. It only differs from 2pl-EL in that the conflicts are handled at commit time in accordance with lazy conflict detection. We omit the evaluation of a 2pl-EE system because it incurs abort rates and performances close to those of 2pl-EL.

- **datm** implements on top of the 2pl-EL base system the commit ordering concurrency algorithm implemented by DATM [38], another TM proposal that implements CS concurrency algorithm in hardware. In this system, (1) conflicts are detected eagerly, (2) transactions commit in the order of their conflicting actions, (3) when a read-after-write conflict is detected, data is forwarded from the writer transaction to the reader transac-
tion, hence, the reader transaction is not stalled, and (4) when a transaction aborts, all the other transactions that it forwarded data to are also aborted. Unlike in DATM that detects conflicts at the word granularity, in our implementation, conflicts are detected at the cache-line granularity for fair comparison with our SONTM. Furthermore, this is an ideal implementation of the concurrency algorithm of DATM. That is, the latencies of control messages that are necessary to keep track of conflicts, race conditions, and data forwarding are not taken into account. We use this version to compare against an ideal version of SONTM to evaluate the impact of using the SON algorithm on performance instead of using another concurrency control algorithm.

- **sontm-ideal** is an ideal version of our SONTM hardware transactional memory system which uses the SON algorithm to provide consistency of shared data. It is ideal in the sense that it does not incur any overheads due to the implementation of the SON algorithm. That is, accessing the write-number table and read histories, updating SON lower and upper bounds, and serializability checks are implemented as zero-cycle operations on top of the base hardware. Therefore, we use this ideal implementation to evaluate the maximum performance gain that can be achieved using the SON algorithm.

- **sontm** is the SONTM hardware TM system described in Chapter 6 with all its features and their associated performance penalties. The size of the virtual table is 64KB. Bits 8-23 are used to map addresses to entries in the table. A 4-entry read history table is kept for past transactions at each processor, in addition to a single entry for the remaining past transactions and the read-set of the active transaction. The read history tables are implemented as perfect bloom filters with no false positives. The SON broadcasts and the read-number responses are implemented similar to memory data responses, i.e. these signals are sent to the directory which forwards them to the target processor.
Table 7.6: Hardware model parameters used for evaluating HTM systems.

<table>
<thead>
<tr>
<th>Component</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processors</td>
<td>8-core CMP, 5 GHz in-order single-issue UltraSparc processors</td>
</tr>
<tr>
<td>Interconnect</td>
<td>packet-switched tiled interconnect consisting of 2 clusters with 4 cores in each</td>
</tr>
<tr>
<td>L1 Cache</td>
<td>based on the MESI protocol, 32 KB private, 4-way, write-back, 1-cycle latency</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>Shared 8 MB, 8-way, 32 banks interleaved with block address, 34-cycles latency</td>
</tr>
<tr>
<td>Directory</td>
<td>on-chip directory with 6-cycles latency</td>
</tr>
<tr>
<td>Main Memory</td>
<td>4 GB, 500-cycles latency</td>
</tr>
<tr>
<td>Virtual Memory</td>
<td>8 KB pages</td>
</tr>
</tbody>
</table>

7.2.2 HTM Experimental Platform

We evaluate the performance of the SON algorithm in hardware (i.e., SONTM) described in Chapter 6 using a simulator of the EL hardware TM system implementation of Bobba et al. [20]. This system provides the base support for TM as described in Section 6.1. It is built using the Simics [26] full-system simulation infrastructure. The Wisconsin GEMS toolset [25] provides support for customizing the memory model. Simics accurately models the Ultra-SPARC architecture, with in-order single-issue processors. The TM support is implemented using magic instructions, i.e., special instructions caught by Simics and passed on to the memory model. The simulated target system runs Solaris 10 to provide OS support for applications.

Since our simulation infrastructure models a directory-based coherence protocol, we implement the directory-based scheme of SONTM described in Section 6.7.

Table 7.6 shows the parameters of our simulations. The simulations take into account at the cycle level the overheads of our SONTM implementation by keeping track of the contention over the interconnect, cache effects, virtual memory, as well as the latencies of accessing the cache, its directory and the memory. These overheads reflect accesses to the virtual write-number table in the main memory, broadcast message latencies, stalls due to NACK responses, and read-number requests and their responses.
7.2.3 HTM Evaluation Metrics

We evaluate the characteristics of the benchmarks and their performances with HTM systems using several performance metrics. These metrics are:

- *The total execution cycles* is the total execution cycles measured from the initialization of the HTM components until the shutdown of these components. Thus, it excludes certain initialization operations such as opening and reading from the input files.

- *The abort rate* is the ratio of the number of aborted transactions to the total number of transactions committed.

- *The wasted cycles* is the total number of cycles spent by transactions that later aborted plus the number of cycles spent for the exponential backoff until restart.

- *The concurrency* is the ratio of the total execution cycles on one thread to the execution cycles of the same benchmark on eight threads.

The above metrics used to evaluate the HTM systems are different than the ones used to evaluate the STM systems described in Section 7.1.4. This is mainly due to the consequences of the different experimental platforms used. First, the STM systems introduce overheads that negatively impact the performance of the benchmark applications. In contrast, the overheads of the HTM systems are significantly lower. Thus, when comparing against the sequential versions of the benchmarks, we measure the slowdown for STM systems whereas we measure the concurrency for HTM systems. Second, the run times of the benchmarks are significantly lower when the benchmarks are run with the STM systems compared to when they are simulated with the HTM systems. Thus, we present speedups of the benchmarks for various number of threads when evaluating the STM systems; however, we evaluate the total execution cycles only on 8 processors to evaluate HTM systems. Finally, the simulation platform used to evaluate the HTM systems allows measuring the wasted cycles with no significant impact on the total execution cycle results. In contrast, measuring wasted cycles for STM systems impacts
the overall performance significantly. Therefore, this metric is not used to evaluate the STM systems.

### 7.2.4 HTM Benchmarks

We evaluate the performance of the HTM systems using the same benchmarks that are used to evaluate the STM systems and that are described in Section 7.1.5.

The running times of our benchmarks differ significantly when they run using the STM systems and when they are simulated using the HTM systems. Thus, we use benchmark parameters different than the ones used to evaluate the STM systems. Nevertheless, we use the same parameters used in earlier work for evaluating other similar HTM implementations [38]. Table 7.7 depicts the parameters given to these benchmarks.

Table 7.8 shows for each benchmark run on the base EL system with 8 threads; the ratio of transactions to the total execution cycles (Xact Ratio), the total number of committed transactions (Xact Count), the average cycles spend for each successful transaction (Xact Cycles), the abort rate (Abort Rate), the ratio of wasted cycles (Wasted Ratio), the average number of read actions per transaction (Xact Reads), the average number of write actions per transaction (Xact Writes),

---

1 A benchmark that runs under one second can take more than a day to simulate using the same input parameters.
Table 7.8: Benchmark characteristics measured with the 2pl-EL HTM system.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Xact Ratio</th>
<th>Xact Count</th>
<th>Xact Cycles</th>
<th>Abort Rate</th>
<th>Wasted Cycles</th>
<th>Xact Reads</th>
<th>Xact Writes</th>
<th>Conc.</th>
</tr>
</thead>
<tbody>
<tr>
<td>list</td>
<td>94.9%</td>
<td>8,192</td>
<td>99,311</td>
<td>0.96</td>
<td>46.0%</td>
<td>2,080.1</td>
<td>0.5</td>
<td>4.01</td>
</tr>
<tr>
<td>bayes</td>
<td>62.7%</td>
<td>797</td>
<td>344,889</td>
<td>1.39</td>
<td>50.7%</td>
<td>10,363.1</td>
<td>1,588.0</td>
<td>2.03</td>
</tr>
<tr>
<td>vacation</td>
<td>56.5%</td>
<td>20,000</td>
<td>5,899</td>
<td>1.01</td>
<td>49.1%</td>
<td>651.3</td>
<td>36.5</td>
<td>1.81</td>
</tr>
<tr>
<td>labyrinth</td>
<td>93.2%</td>
<td>112</td>
<td>923,514</td>
<td>5.89</td>
<td>88.2%</td>
<td>88,451.5</td>
<td>11,356.7</td>
<td>0.41</td>
</tr>
<tr>
<td>yada</td>
<td>99.4%</td>
<td>5,080</td>
<td>26,430</td>
<td>4.96</td>
<td>94.5%</td>
<td>1,269.5</td>
<td>222.8</td>
<td>0.46</td>
</tr>
<tr>
<td>intruder</td>
<td>57.0%</td>
<td>11,267</td>
<td>834</td>
<td>0.31</td>
<td>33.7%</td>
<td>28.5</td>
<td>6.5</td>
<td>4.92</td>
</tr>
<tr>
<td>genome</td>
<td>74.2%</td>
<td>10,785</td>
<td>3,040</td>
<td>0.08</td>
<td>18.0%</td>
<td>177.1</td>
<td>5.9</td>
<td>5.24</td>
</tr>
<tr>
<td>kmeans</td>
<td>11.5%</td>
<td>43,706</td>
<td>396</td>
<td>0.06</td>
<td>2.3%</td>
<td>40.0</td>
<td>19.0</td>
<td>5.10</td>
</tr>
<tr>
<td>ssca2</td>
<td>7.8%</td>
<td>47,277</td>
<td>298</td>
<td>0.01</td>
<td>0.4%</td>
<td>3.0</td>
<td>2.0</td>
<td>4.47</td>
</tr>
</tbody>
</table>

(Xact Writes) and concurrency (Conc.). From the table, it can be seen that list, bayes and labyrinth have longer transactions (i.e., 100K or more cycles are spent on average for successful transactions); list, bayes, vacation, labyrinth and yada have higher abort rates (around 1.0 and above) and, hence, higher wasted cycles ratios. Figure 7.6 depicts the abort rate and Xact Cycles distribution of each benchmark.

The results shown in the figure generally correlate well with the results presented for the STM systems shown in Figure 7.1. However, there are also differences. These differences are caused by the following factors. First, different parameters are used to evaluate the HTM systems and the STM systems as described above. Second, the STM systems incur much higher overheads compared to the HTM systems, which causes transactions to have different characteristics. Third, with the HTM systems, every memory access made within the transactions are considered as transactional accesses. In contrast, with the STM systems, only the shared variable accesses marked by the programmer or the compiler are considered as transactional accesses. This leads to different transaction overheads, thus, transaction sizes. Finally, HTM systems are subject to false sharing, i.e. transactions may conflict when they access different words within the same cache block. This impacts the abort rates in HTM systems. In contrast, false sharing is not a factor in STM systems since transactional accesses are kept track of at the word granularity.
Figure 7.6: Abort rate and transaction cycles distribution of the benchmarks measured with the 2pl-EL HTM system.

7.2.5 Experimental Results

The impact of the SON Support

Figure 7.7(a) shows the total execution cycles of each HTM system for each benchmark normalized with respect to the total execution cycles of the 2pl-EL HTM system. A number of observations can be made from the figure.

- The total execution cycles of HTM systems correlate with the abort rates shown in Figure 7.7(b) and with the wasted cycles shown in Figure 7.7(c). This shows that reducing the abort rates of a benchmark positively impact its performance by reducing the cycles wasted for aborted transactions.

- SONTM implementations (i.e., sontm-ideal, and sontm) outperform the 2pl-EL for the benchmarks with high abort rates (i.e., list, bayes, vacation and labyrinth). The maximum gain is 70% which is achieved with sontm-ideal for vacation. This result is expected since abort rates are significantly reduced for these benchmarks.
For yada, the performance of 2pl-EL is significantly lower compared to the performance of the other implementations due to the live-locking. That is, transactions are repeatedly aborting each other without making forward progress until the exponential back-off mechanism breaks the conflict pattern. For intruder, the performance of sontm-ideal is 21% better than the performance of the 2pl-EL because the abort rate is reduced to 0.16 from 0.31. For genome, a small gain in performance is achieved because the abort rate is reduced from 0.08 to 0.02 with CS techniques. For kmeans, the performance does not improve with SONTM implementations because most transactions in this benchmark are not conflict-serializable. For ssca2, the performance of sontm-ideal is equivalent to the performance 2pl-EL because this benchmark contains very short transactions that are not conflict-serializable.

The performance of the sontm-ideal system matches or slightly outperforms that of the datm system for the benchmarks evaluated. This shows that the SON algorithm is an efficient method for implementing CS in a hardware TM system. In fact, in some benchmarks (intruder, genome, kmeans and ssca2), sontm-ideal slightly outperforms datm due to stalls incurred by the commit ordering of transactions.

The difference in performance between sontm-ideal and sontm is small (between 1% and 4%) for applications with long transactions that perform a large number of transactional read and write actions (list, bayes, vacation, and labyrinth). This shows that the optimization described in Section 6.5 is successful in eliminating the overheads of sontm caused by the loading of write-numbers from the virtual write-number table during transactional actions.

The difference in performance between sontm-ideal and sontm is moderate (up to 21%) for applications with short transactions (intruder, genome, kmeans, ssca2). This is due the constant overheads incurred by the sontm implementation on the commit operations of transactions. As it was mentioned in Section 6.6, these overheads consist of
serialization of commit operations, broadcasting SONs at commit time, stalling commit operations until the SON broadcast messages are received by every other processor, and storing write-numbers into the virtual write-number table. Thus, it can be concluded that impact of such hardware operations on performance is negligible for long transactions; however they impact the performance of short transactions. For instance, the commit operation of \textit{ssca2} only consists of committing on average two addresses into memory when the application is run with \textit{2pl-EL}. Thus, the extra overheads of \textit{sontm} increases the commit times of such transactions with short commits, leading to performance drops. Further, longer commit times also increase the chances of active transactions conflicting with committing transactions, which lead to additional aborts.

- \textit{2pl-LL} outperforms \textit{2pl-EL} on average and in most benchmarks (except in \textit{bayes, intruder, genome} and \textit{ssca2}) due to its lower abort rates. However, the performance of \textit{2pl-LL} is still significantly lower than that of \textit{sontm} because of the strict 2PL concurrency algorithm it implements.

- For \textit{ssca2}, we observe that the performance of \textit{ssca2} is reduced by 14\% and the performance of \textit{kmeans} is reduced by 6\% when it is run with \textit{sontm} compared to when they are run with \textit{2pl-EL}. This is because these benchmarks present the worst case scenario for \textit{sontm}; a large number of very short transactions are executed and the abort rates are close to zero. Thus, no performance gain is obtained with the relaxed concurrency of CS, yet the overheads of \textit{sontm} mentioned above lower performance. These benchmarks can benefit from an adaptive concurrency algorithm technique, as was discussed in Section 6.3. That is, a simple algorithm can detect that these benchmarks only consists of short transactions that rarely conflict and switch the concurrency algorithm to 2PL by disabling the SONTM features similar to the adaptive scheme described in Section 5.8.
Figure 7.7: Normalized total execution cycles and abort rates on 8 processors.
Figure 7.7(b) shows the abort rates of each HTM system evaluated for the benchmarks evaluated. The figure shows that the abort rates are high for list, bayes, vacation, labyrinth and yada when they are run with 2pl-EL. However, the sontm-ideal and datm implementations reduce abort rates significantly in these benchmarks. For instance, the abort rate of the list benchmark is reduced from 0.96 with 2pl-EL to 0.01 with datm and to 0.15 by sontm-ideal. This result confirms earlier work [3, 4, 38] that CS reduces abort rates in TM systems. Further, the figure shows that the abort rates of the SONTM implementations (sontm-ideal and sontm) are also small and significantly lower than that of 2pl-EL. This demonstrates that abort rates can also be reduced significantly by implementing the SON method as opposed to implementing commit ordering algorithm of datm. Also, the difference in abort rates between sontm-ideal and sontm is generally small (at most 0.09) except for bayes. This leads us to conclude that impact of the size of write-number table and the aliasing it causes is not significant. For bayes, aliasing over the write-numbers table impact the performance as this benchmark has long transactions with high contention. The abort rates of kmeans and ssca2 are already low when they are run with 2pl-EL. Thus, we do not observe any improvement when these benchmarks are run under CS. The abort rate of intruder is reduced from 0.31 to 0.16 with sontm-ideal and to 0.23 with sontm. The genome benchmark has a low abort rate of 0.08 with 2pl-EL and this rate is reduced to 0.02 with sontm-ideal and sontm. Finally, 2pl-LL incurs lower abort rates compared to 2pl-EL due to the lazy handling of conflicts. However, the abort rates of 2pl-LL are still significantly higher than the abort rates of the systems that implement CS.

Figure 7.7(c) shows the wasted cycles normalized with respect to the total execution cycles of 2pl-EL for the benchmarks evaluated. Due to high contention and long-running transactions, wasted cycles are high for list, bayes, vacation, labyrinth, yada and intruder. We also see that implementing CS instead of 2PL reduces the wasted cycles. This directly correlates with the reduction in the abort rates shown in Figure 7.7(b).
Table 7.9: Normalized execution breakdown on 8 processors.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>HTM</th>
<th>non-trans</th>
<th>trans</th>
<th>wasted</th>
<th>commit</th>
<th>backoff</th>
<th>barrier</th>
<th>stall</th>
</tr>
</thead>
<tbody>
<tr>
<td>list</td>
<td>2pl-EL</td>
<td>3.04</td>
<td>48.89</td>
<td>45.05</td>
<td>0.00</td>
<td>0.96</td>
<td>2.07</td>
<td>0.00</td>
</tr>
<tr>
<td></td>
<td>sontm</td>
<td>3.78</td>
<td>50.17</td>
<td>7.88</td>
<td>0.03</td>
<td>0.08</td>
<td>4.53</td>
<td>0.00</td>
</tr>
<tr>
<td>bayes</td>
<td>2pl-EL</td>
<td>0.69</td>
<td>10.98</td>
<td>49.89</td>
<td>1.00</td>
<td>0.77</td>
<td>36.12</td>
<td>0.54</td>
</tr>
<tr>
<td></td>
<td>sontm</td>
<td>0.79</td>
<td>8.83</td>
<td>32.64</td>
<td>0.34</td>
<td>0.17</td>
<td>25.45</td>
<td>0.10</td>
</tr>
<tr>
<td>vacation</td>
<td>2pl-EL</td>
<td>7.37</td>
<td>14.41</td>
<td>10.45</td>
<td>0.03</td>
<td>38.64</td>
<td>31.59</td>
<td>0.00</td>
</tr>
<tr>
<td></td>
<td>sontm</td>
<td>7.63</td>
<td>16.22</td>
<td>3.53</td>
<td>0.29</td>
<td>0.96</td>
<td>9.39</td>
<td>0.17</td>
</tr>
<tr>
<td>labyrinth</td>
<td>2pl-EL</td>
<td>0.40</td>
<td>4.96</td>
<td>85.12</td>
<td>0.03</td>
<td>3.05</td>
<td>6.44</td>
<td>0.00</td>
</tr>
<tr>
<td></td>
<td>sontm</td>
<td>0.47</td>
<td>5.23</td>
<td>31.35</td>
<td>0.03</td>
<td>0.06</td>
<td>5.83</td>
<td>0.00</td>
</tr>
<tr>
<td>yada</td>
<td>2pl-EL</td>
<td>0.10</td>
<td>4.93</td>
<td>45.13</td>
<td>0.01</td>
<td>49.38</td>
<td>0.46</td>
<td>0.00</td>
</tr>
<tr>
<td></td>
<td>sontm</td>
<td>0.17</td>
<td>3.67</td>
<td>2.82</td>
<td>0.04</td>
<td>0.07</td>
<td>0.12</td>
<td>0.03</td>
</tr>
<tr>
<td>intruder</td>
<td>2pl-EL</td>
<td>41.80</td>
<td>23.29</td>
<td>18.14</td>
<td>0.09</td>
<td>15.51</td>
<td>1.17</td>
<td>0.01</td>
</tr>
<tr>
<td></td>
<td>sontm</td>
<td>42.32</td>
<td>26.26</td>
<td>16.18</td>
<td>2.23</td>
<td>9.83</td>
<td>1.22</td>
<td>2.25</td>
</tr>
<tr>
<td>genome</td>
<td>2pl-EL</td>
<td>8.71</td>
<td>56.13</td>
<td>10.34</td>
<td>0.05</td>
<td>7.64</td>
<td>17.12</td>
<td>0.00</td>
</tr>
<tr>
<td></td>
<td>sontm</td>
<td>16.52</td>
<td>57.68</td>
<td>3.56</td>
<td>1.26</td>
<td>0.48</td>
<td>16.80</td>
<td>0.80</td>
</tr>
<tr>
<td>kmeans</td>
<td>2pl-EL</td>
<td>87.91</td>
<td>9.05</td>
<td>0.65</td>
<td>0.11</td>
<td>1.65</td>
<td>0.60</td>
<td>0.03</td>
</tr>
<tr>
<td></td>
<td>sontm</td>
<td>89.35</td>
<td>9.79</td>
<td>2.37</td>
<td>1.07</td>
<td>2.11</td>
<td>0.68</td>
<td>0.27</td>
</tr>
<tr>
<td>ssca2</td>
<td>2pl-EL</td>
<td>69.47</td>
<td>8.07</td>
<td>0.09</td>
<td>0.13</td>
<td>0.32</td>
<td>21.92</td>
<td>0.00</td>
</tr>
<tr>
<td></td>
<td>sontm</td>
<td>71.15</td>
<td>10.34</td>
<td>0.90</td>
<td>4.06</td>
<td>0.51</td>
<td>20.78</td>
<td>6.28</td>
</tr>
</tbody>
</table>

The breakdown of execution

Table 7.9 shows the breakdown execution of benchmarks when they are run with 2pl-EL and sontm on 8 processors. The average number cycles spent on every phase of execution by each processor is normalized with respect to the total execution cycles of the benchmark when it is run with 2pl-EL on 8 processors. The execution is broken down to the following phases.

- non-trans: The number of cycles spent when no transactions are running.
- trans: The number of cycles spent when an active transaction is running.
- wasted: The number of cycles spent for transactions that later aborted.
- commit: The number of cycles spent for commit operations.
- backoff: The number of cycles spent for exponential backoff.
barrier: The number of cycles spent at the synchronization barriers before and after the execution of the kernel (excludes initializations, allocations, etc.).

stall: The number of cycles spent waiting to acquire the commit token.

The following observations can be made from the table. First, \texttt{sontm} reduces the wasted cycles in \texttt{list}, \texttt{bayes}, \texttt{vacation}, \texttt{labyrinth}, \texttt{yada} and \texttt{genome}. Further, it reduces execution cycles spent for exponential backoff in \texttt{vacation}, \texttt{yada}, \texttt{intruder}, and \texttt{genome}. These reductions are direct consequence of lower abort rates shown in Figure 7.7(b). Second, some improvement is also obtained in the cycles spent for barriers. This is again due to the reduction in abort rates which allows transactions to reach the final barrier sooner. Third, the number of cycles spent outside transactions is increased for \texttt{genome}. This is due to the increased cache misses caused by the write-numbers. This eliminates the gains obtained through the reduction of abort rates in this benchmark. Fourth, for \texttt{ssca2}, the cycles spent for commit and stall components are increased with \texttt{sontm}. This is due to the extra overheads and the serialization of commit operations in \texttt{sontm} as discussed in Section 6.6. As a result, the performance of \texttt{ssca2} is overall reduced by 14\% as shown in Figure 7.7(a).

The impact of SONTM parameters

Figure 7.8(a) shows the total execution cycles of \texttt{sontm} normalized with respect to the total execution cycles of \texttt{2pl-EL} for the benchmarks. In each experiment, a different range of address bits is used for hashing. The size of the write-numbers table is constant at 64K. Since the cache lines are 64 bytes long, the 6 least significant address bits are not used to index the virtual write-number table. It can be seen that the address bits selection impacts the performance in some benchmarks. This impact may be as large as 31\% of total execution cycles in \texttt{list}, or as small as 3\% in \texttt{vacation}. This mostly depends on how the data structures are aligned with cache lines in benchmarks. However, it is possible to use address bits starting from the 6th, 8th or 10th bits and generally produce good performance.
Figure 7.8(b) shows the total execution cycles of *sontm* normalized with respect to the total execution cycles of 2pl-EL for the benchmarks. We simulate execution with write-numbers tables of sizes 4KB, 16KB, 32KB, 48KB and 64KB. In each experiment, the 16 address bits starting from the 8th bit are used to hash into the table. It can be seen that the impact of the size of the table is small except for *bayes*. This is because transactions in *bayes* access a large number of shared addresses requiring a larger write-numbers table to eliminate the impact of aliasing. Nonetheless, the table size of 64K is generally adequate.

Figure 7.8: The impact of selected address bits and the size of the virtual table.
Summary of the Results

The experimental results show that the SON algorithm is as effective as the commit ordering algorithm of DATM in improving the performance by reducing the abort rates. This performance improvement is more significant for applications with long transactions and high abort rates (such as list, bayes, vacation, labyrinth and yada) because for these applications the impact of reducing the abort rates outweigh the overheads of testing for CS. In contrast, for applications with short transactions and low abort rates (such as intruder, genome, kmeans and scca2), the overheads eliminate the performance gains of CS. However, performance is reduced only for one benchmark (i.e., scca2) by 14%. An adaptive algorithm can be used for such applications to switch to the 2PL concurrency control algorithm that performs better, thus avoiding the extra overheads of testing for CS. Finally, the results show that the parameters of SONTM do not impact the performance significantly and parameters that work well for a wide range of applications can be found.
Chapter 8

Related Work

This chapter discusses previous research in databases and in transactional memory that are related to the SON algorithm and its software and hardware implementations described in earlier chapters.

8.1 Transaction Management in Databases

Transaction management is a well-studied field of research in database management systems (DBMSs). Even though some systems implement relaxed concurrency algorithms in order to increase concurrency, serializability is generally considered as the main requirement to preserve the correctness of database state. Serializability is implemented using different concurrency control protocols such as: two-phase-locking, timestamp ordering, commit ordering, and multi-version concurrency control [9]. Timestamp ordering and optimistic concurrency protocols are the most related to our work and, hence, are the focus of the remainder of this section.

Timestamp ordering assigns each transaction a timestamp value which represents the serialization order of that transaction. Further, each database element is associated with a read time and a write time. The read time of a database element is the highest timestamp of a transaction that has read the element; similarly, the write time is the highest timestamp of a transaction that
has written to the particular element. Each transaction is assigned a fixed timestamp at start time using a global timestamp clock. Every time a transaction accesses a database element, the timestamp of the transaction is compared against the read time and/or the write time of the element. This allows transactions to serialize in the order of their start times. More specifically, when a transaction reads an element, its timestamp must be higher than the write time of the element; otherwise, the transaction aborts. Similarly, when a transaction writes to an element, its timestamp must be higher than both the write time and the read time of the element. After each read or write action, transactions update the read and write times of the elements they access.

The SON algorithm is similar to timestamp ordering, but there exist key differences. Mainly, the timestamp algorithm assigns each transaction a fixed timestamp at start time, whereas our SON algorithm selects an SON at commit time based on the conflicts of transactions. This causes the timestamp algorithm to serialize transactions only in the order of their start times, whereas the SON algorithm is more flexible for serializing transactions, hence, it allows better concurrency.

Other concurrency control protocols have also been proposed for DBMSs. Optimistic concurrency control (OCC) protocols [21] are particularly similar to lazy VM HTMs and obstruction-free STMs in that they are non-blocking and serialization is achieved at commit time. That is, transactions are allowed to execute unhindered until they commit, at which point they are validated. Thus, the execution of transactions consists of a read phase, a validation phase and a write phase. During the read phase, transactions read database elements and their writes are buffered. During validation, transactions are verified for serializability. Finally, during the write phase transactions update the database elements. We consider specific algorithms of OCC similar to our work, hence, discuss these algorithms in the remainder of this section.

Lam et al. [23] proposes the OCC-DA protocol which improves upon the timestamp ordering protocol by dynamically adjusting timestamps in order to reduce the rate of aborts. Timestamps of transactions are dynamically adjusted depending on the type of the conflicts. For this purpose, they introduce forward adjustment and backward adjustment rules. Lindstrom
and Raatikainen [24] build upon the OCC-DA with their OCC-DATI protocol. In OCC-DATI, transactions are assigned timestamp intervals which represent serializability ranges of transactions. Further, database elements are assigned read-timestamps and write-timestamps that denote the largest timestamp of committed transactions that have read or written, respectively, a database element. The timestamp intervals of transactions are adjusted at commit time using the backward adjustment and forward adjustment rules according to the conflicts detected. A transaction aborts if its timestamp interval becomes empty.

The SON algorithm is similar to OCC-DATI mainly in two aspects. First, timestamps are dynamically adjusted at commit time based on the conflicts of transactions. Second, forward and backward adjustment rules are used to adjust the timestamps of transactions. These rules are analogous to the forward-serialization and backward-serialization rules described in Section 4.1.

Our SON algorithm differ from OCC-DATI in two regards. First, OCC-DATI validates the transactional reads at commit time, whereas our SON algorithm validates the reads when the read action takes place by updating the SON lower bound of transactions. This allows the SON algorithm to avoid inconsistent reads and to abort non-serializable transactions earlier without waiting until commit time. Second, in OCC-DATI, valid transactions are assigned their final timestamps using a global timestamp counter. In our SON algorithm, the final SON value is assigned as the lower bound plus the number of threads. This approach leads to simpler implementation and reduces the likelihood of overflowing SON bounds.

8.2 Conflict-Serializability in Transactional Memory

Our work is the first to consider the use of CS for concurrency control in TM systems. Our initial results on the implementation of TSTM was followed by Ramadan et al. [38] who propose the DATM hardware TM system that implements CS using a commit ordering concurrency control algorithm. In other words, transactions in DATM commit in the order of their conflicts;
if a transaction TX2 writes to an item already read by another active transaction TX1, TX2 stalls its commit until TX1 commits. Since DATM enforces commit ordering, in order not to stall read actions, read-after-write dependences are handled by forwarding data from the active writer transactions to the readers. This requires DATM to implement a novel cache coherence protocol to keep track of dependences and forwarded data.

Even though DATM and our SONTM both implement CS and hence provide the same level of concurrency, our SONTM implementation is simpler and more appealing than DATM in a number of ways.

First, DATM requires a novel cache coherence protocol, which results in a complex hardware implementation. Given that future HTM systems will likely be based on existing coherence protocols, SONTM scheme presents a clear advantage. SONTM requires no changes to existing cache coherence protocols and only the addition of simple hardware components, such as conflict flags and read histories.

Second, transactions in DATM can read uncommitted data due the data forwarding scheme. This may cause active transactions to enter inconsistent state leading to problems such as infinite loops, exceptions, and data corruption. Operating system support is necessary to properly handle these problems and to make sure that the consistency of system state is preserved. SONTM, on the other hand, does not allow inconsistent state; transactions are immediately aborted if they perform inconsistent reads.

Third, the data forwarding scheme of DATM also leads to cascading aborts. That is, a transaction must be aborted if it receives forwarded data from another transaction that later aborts. This may increase the rate of aborts and also requires additional hardware support as all the transactions must be aware of aborted transactions.

Fourth, in DATM, each transaction must keep track of all the conflicts among all the transactions so that it can commit in the order of its conflicts. For this purpose, DATM relies on a bus-based cache coherence protocol where all the conflicts are seen by all the transactions. Bus-based protocols are generally not appealing because they do not scale well with increasing
number of processors. Implementing DATM on top of a directory-based protocol would further complicate its hardware design due to the extra messages that need to be sent every time a conflict is detected. Our SONTM implementation, on the other hand, does not require each transaction to have the complete information of conflicts; each transaction keeps tracks of only its own conflicts. Thus, we easily implement SONTM on top of a directory-based protocol with only the addition of broadcast messages that are only sent when transactions commit.

Finally, DATM implements word-level conflict detection which is not common in traditional TM implementations. We use cache-line level conflict detection in SONTM, even though word-level conflict detection can easily be adapted.

Ramadan et al. [39] also implement the concurrency algorithm of DATM in their software system called DASTM. DASTM, like DATM, commits transactions in the order of their conflicts, forwards data in case of read-after-write conflicts, and admits cascading aborts (i.e., aborting transactions that receive forwarded data). Thus, DASTM also allows transactions to read inconsistent or invalid data due to data forwarding, causing transactions to enter infinite loops, write to incorrect addresses, or fail program assertions. For this purpose, DASTM implements special mechanisms that track inconsistent reads and infinite loops, and restart transactions with no data forwarding support. This approach increases overheads and abort rates. In addition, DASTM implementation requires an expensive read validation phase at commit time and acquires a global lock during each transactional action, leading to further overheads that do not exist in our TSTM implementation.

### 8.3 Other Approaches in TM Systems

Riegel et al. [41, 40] investigate the use of snapshot isolation which is a more relaxed consistency model than serializability. Snapshot isolation reduces the abort rates significantly by allowing transactions to access old versions of shared data when there exist conflicts over the new versions. However, snapshot isolation does not provide serializability and may lead to
incorrect execution in some applications. Thus, it requires the programmer to make sure that it is applicable or to transform some read accesses to write accesses for correctness. If serializability is required, snapshot isolation again resorts to 2PL to ensure correctness. In contrast, our work reduces the abort rates by implementing the CS model, and does not require any programmer intervention to ensure correctness.

Fernandes et al. [13] implement Multi-Version Concurrency Control (MVCC) in their JVSTM system. MVCC is a relaxed concurrency control algorithm that stores multiple versions of each shared data item in the order they were committed by transactions. This allows transactions to read earlier versions of shared data items, hence, avoid aborting if reading the latest version of a shared data item prevents serializing the transaction. This approach improves the performance of the read-only transactions. Although this concurrency control algorithm is similar to our SON algorithm, it still serializes reader-writer transactions in the order they commit without adjusting the serialization order. Thus, it cannot provide an efficient method for improving the performance of long transactions with high contention.

Similarly, Manassiev et al. [27] also use MVCC at the page level for implementing a distributed shared memory system on clusters. That is, their system allows multiple replicas (i.e. versions) of the same page to exist at different nodes. When a read-only transaction receives an update for a page in its read-set, it delays applying the update until it commits, hence, reading from a consistent snapshot of the memory space. Thus, this system also implements a relaxed concurrency control algorithm which allows reader transactions (and only reader transactions) to be serialized with the conflicting writer transactions without aborting the reader transaction. However, in contrast to our work, this system cannot serialize transactions that conflict due to updates on the same data items. Thus, although this concurrency control algorithm reduces abort rates for read-only transactions, it is more strict than our algorithm because it causes aborts when there are conflicting writer transactions. In addition, our TM implementation differs from this system in that it targets shared-memory systems at the finer word and cache level granularity.
TL2 [12] and RSTM [31] implement a technique called global versioned clock to reduce the overheads of read-set validation. With this technique, each writer transaction updates a global clock when it commits. Transactions read this clock when they start executing. Thus, a transaction can easily verify that the shared items it read are still up to date by checking the current value of the global clock. If the value of the global clock has not changed, all the items in the read-set are guaranteed not to have changed. Further, Olszewski et al. implement the JudoSTM [36] that implements value-based conflict detection. With this approach, conflicts are checked by comparing the old and new values of the shared items. That is, if the value of a shared item has not changed, any updates on this item is ignored reducing the number conflicts detected. All these techniques described above improve the performance of the STM systems by reducing the overheads or the number of conflicts. However, in contrast to our proposed SON algorithm, they are optimization techniques rather than relaxed concurrency control algorithms. That is, these systems still implement 2PL in that they abort or delay transactions every time a conflict is detected.

Calstrom et al. [5] assert that long-running transactions are important for the ease of parallel programming and recognize that the use of 2PL limits concurrency. They propose transactional collection classes to reduce the number of conflicts and demonstrate its use for some example data-structures. They manually determine which conflicts must abort transactions and which conflicts should not. However, their approach requires knowledge of the semantics of data structures and the dependencies that exist. In contrast, our work provides a means for reducing conflicts in the same type of applications without programmer intervention.

Shriraman et al. [48] implement FlexTM which uses software mechanisms to decide on the policies (eager or lazy) that will be used to handle conflicts. They show that using lazy version management leads to fewer aborts and hence to more concurrency. However, similar to other existing HTM systems, lazy version management of FlexTM is also based on the 2PL concurrency algorithm which is more strict than the SON algorithm of our SONTM.

Herlihy et al. [18] and Skare et al. [49] propose the early-release technique and successfully
use it to reduce the number of conflicts in applications that use linear data structures, such as linked-lists. However, the use of early-release requires that the programmer be aware of the semantics of the application and ensure that accesses to the particular address do not affect the serializability of transactions.

Ni et al. [35] and Calstrom et al. [6] propose open nested transactions as a mean of reducing conflicts for long-running transactions. With open-nesting, the programmer divides a transaction into a series of shorter transactions to avoid conflicts. However, this technique also requires knowledge of the application semantics and, thus, increases programmer effort.
Chapter 9

Conclusions

Transactional Memory (TM) [19] systems have gained considerable popularity in recent years [12, 22, 48]. This is mainly because TM systems promise to ease parallel programming by eliminating the need for user-locks. Performance close to that of fine-grain use of locks can be achieved with the programming simplicity of coarse-grain use of locks, i.e., by just identifying the critical sections and the shared data accesses.

To date, significant improvements in TM performance and functionality have been achieved. However, researchers have mostly concentrated on applications with short-running transactions and low contention [12, 31, 22]. They were able to deliver good performance for these applications by providing fast transactional operations. For this purpose, they relied on a simple concurrency algorithm called 2-Phase Locking (2PL), which aborts transactions every time a conflict is detected. 2PL is suitable for applications with short transactions and low contention because for these applications the cost of aborting and restarting transactions is low.

However, 2PL fails to deliver good performance for applications with long-running transactions and high contention due to excessive aborts which limit concurrency and performance. Consequently, one major outstanding problem for TM systems has been providing adequate performance for this type applications.

We propose to use an algorithm more relaxed than 2PL for enforcing serializability in TM
systems. This algorithm is based on a special type of serializability called conflict-serializability (CS) [9], which allows transactions to commit even when they make conflicting accesses. This results in lower abort rates. The algorithm requires keeping track of conflicts among transactions and it serially orders transactions as they execute by assigning them *Serializability Order Numbers (SONs)* based on the order of their conflicting actions. We formulated the operations of the SON algorithm and showed that transactions that can be assigned SONs are serializable, and hence, result in correct execution and consistent system state.

We showed that the SON algorithm can be implemented both in software and in hardware on top of standard TM implementations. Our software implementation is based on a eager-acquire software base system with visible reads. This configuration results in the most efficient implementation, although implementation is also possible for other configurations. Our software implementation keeps track of the conflicting transactions by keeping additional metadata in memory which are updated when transactions commit. Thus, the abort rates are reduced at the expense of extra cache misses and higher latency commit operations.

The hardware implementation is based on an EL base hardware TM system and it requires little additional hardware complexity. This additional hardware consists of a few simple components per processor: a small read history table, a set of registers to hold order and related data, and a set of flags for keeping track of conflicts. In addition, a table is maintained in virtual memory to keep track of the SONs of writer transactions for each shared data address. The use of these components introduces some overheads that reflect cache misses due to the accesses to the virtual table and messages used for communicating SONs, commit events and conflicts among transactions. The simplicity of our approach stands in contrast to other approaches that involve more complex changes to hardware, in particular cache coherence protocols [38].

We evaluated the impact of implementing CS on performance using standard TM benchmarks. Our evaluation shows that the use of CS does indeed result in significantly lower abort rates compared to 2PL, leading to better performance for applications that have long transactions and high contention. For example, the use of CS improves the throughput of transactions
in a linked-list benchmark by 114% and in another benchmark (*vacation*) by 199% on 16 processors compared to 2PL on the same number of processors. We also compare the performance of our system to that of a blocking STM system (TL2) and show that the throughput of transactions on our system exceeds that of TL2 in applications with long transactions and high abort rates.

The evaluation of our HTM system uses a full system simulator and it shows that our HTM system outperforms the base HTM system that uses 2PL. Performance improves by 29% on average over nine benchmarks. Further, the performance of our system is only 6% below the performance of an ideal CS implementation that incurs no overheads, which leads us to believe that our implementation is efficient. We also compare against the performance of the concurrency control protocol used by another TM system (DATM) that implements CS and show that its performance is comparable to that of ours. However, our system achieves this performance with significantly less hardware complexity.

Our work can be extended in a number of ways. First, other relaxed concurrency control algorithms (such as timestamp ordering, commit ordering, and multi-version concurrency control) can be used in software or in hardware TM systems to implement the CS model. These algorithms differ in terms of the concurrency they allow and in terms of their implementation overheads. Thus, a wider range of algorithms can be chosen depending on the characteristics of the target application. For instance, more relaxed algorithms can be used for applications with higher contention, whereas stronger algorithms can be preferred for applications with low contention.

Second, similar to our software system, our hardware TM system can also be extended to address the overheads for applications that have low contention and short transactions such as ssca2, by using adaptive techniques. A simple algorithm can detect that an application only consists of short transactions that rarely conflict and switch the concurrency algorithm to 2PL.

Third, our systems can be extended to support nested transactions. This may require analyzing the impact of nested transactions on consistency, possibly defining new semantics for
nested transactions, and designing the necessary data structures or the hardware components that implements them.

Finally, the SON algorithm can be extended to assign transactions SON ranges instead of final SON values. This can improve the concurrency by giving transactions more flexibility for serialization. However, it may also result in more complex software and hardware implementations as it increases the amount of bookkeeping required.
Bibliography


