Increasing the Scalability of a Software Transactional Memory System

Faustino Dabraio da Silva

Dissertation submitted to obtain the Master Degree in

Information Systems and Computer Engineering

Jury

Chairman: Prof. Nuno João Neves Mamede
Supervisor: Prof. João Carlos Serrenho Dias Pereira
Co-Supervisor: Prof. João Manuel Pinheiro Cachopo
Members: Prof. João Pedro Faria Mendonça Barreto

May 2012
Acknowledgements

First and foremost, I am grateful for the support and orientation of my tutor, Prof. João Pereira, who for more than a year was always available every week to guide me through the whole process, always giving me helpful ideas and steering me in the right direction.

Also, this thesis would not have been possible without the help of Prof. João Cachopo and Prof. Sérgio Fernandes, who took the time to give me invaluable insight on how the JVSTM works, as well as suggestions on how it could be improved.

Finally, a word of gratitude to my family and friends, who supported me the whole time and provided me with much needed motivation to reach my goals.

Lisbon, May 14, 2012
Faustino Dabraio da Silva
Resumo

A Memória Transaccional em Software (STM) introduz na programação o modelo transaccional conhecido dos sistemas de bases de dados, com o objectivo de fornecer uma alternativa mais simples e flexível aos tradicionais trincos usados em programação concorrente. Apesar de as propostas de STM existentes cumprirem as promessas de simplicidade e flexibilidade, ainda há margem para melhorias em termos de performance.

Neste trabalho comecei por analisar algumas STMs existentes para perceber o seu funcionamento e compreender as diferentes opções de desenho que existem. Depois de analisar em detalhe o funcionamento da Java Versioned Software Transactional Memory (JVSTM), identifiquei possíveis melhorias para os seus algoritmos e estruturas de dados, implementei-as e avaliei se (e porquê) tiveram o impacto esperado na performance da JVSTM.
Abstract

Software Transactional Memory (STM) introduces the transactional model of database systems to software programming, with the aim of being a simpler and more modular alternative to locks in concurrent programming. While most existing STM engine proposals are successful in delivering the promises of increased simplicity and modularity, there is room for improvement in terms of performance.

In this work I started by analyzing existing STMs to understand how they work and what different design alternatives exist. Then, after analyzing how the Java Versioned Software Transactional Memory (JVSTM) works, I identified potential improvements to its key data structures and algorithms, implemented them and evaluated whether or not (and why) they had the intended impact on JVSTM’s performance.
Palavras-Chave

*Palavras-Chave*

Memória Transaccional em Software
Escalabilidade
Programação Concorrente com Objectos

*Keywords*

Software Transactional Memory
Scalability
Concurrent Object-Oriented Programming
# Contents

1 Introduction 1

1.1 Motivation 1

1.2 Context 2

1.3 Goals 2

1.4 Outline 3

2 Software Transactional Memory 5

2.1 Basic Concepts of STMs 5

2.1.1 Transactions 5

2.1.2 Transactional Memory 6

2.1.3 Transaction Nesting 6

2.1.4 Read and Write Sets 7

2.1.5 In-place/Out-of-place Updates 7

2.1.6 Contention Management 7

2.1.7 Long-running Transactions 7

2.2 Design Alternatives 7

2.2.1 Synchronization Technique 8

2.2.2 Transactional Data Granularity 9

2.2.3 Concurrency Control 9

2.2.4 Lock Acquisition and Recovery Strategy 10

2.2.5 Lock Placement 10

2.2.6 Transaction Validation 10

3 Related Work 13

3.1 DSTM 13

3.2 McRT-STM 15

3.3 TL2 16
3.3.1 CS-STM ................................................. 17
3.4 A Lock-based Protocol for STM (LBP-STM) ......................... 19
3.5 Summary and Conclusions ........................................ 20

4 JVSTM ..................................................... 23
4.1 The Versioned Memory Model ....................................... 24
4.2 Transaction Lifecycle ................................................ 24
   4.2.1 Begin .................................................. 25
   4.2.2 Execution .............................................. 25
   4.2.3 Validation and Commit ................................... 25
   4.2.4 Abort .................................................. 26
   4.2.5 Nested Transactions .................................... 27
4.3 Important Data Structures .......................................... 27
   4.3.1 VBox .................................................... 27
   4.3.2 Read Set ............................................... 28
   4.3.3 Write Set ............................................... 29
      4.3.3.1 During Execution .................................. 29
      4.3.3.2 After Execution ................................... 30
   4.3.4 ActiveTransactionsRecord ................................ 32
4.4 Performance Evaluation ............................................ 33
   4.4.1 Statistical Data Collection .................................. 33
   4.4.2 Test Environment ....................................... 34
      4.4.2.1 Hardware ......................................... 34
      4.4.2.2 Benchmark ....................................... 34
   4.4.3 Early Conclusions ....................................... 35

5 Proposed Optimizations .............................................. 37
5.1 Implement the Read Set Using an Identity Hash Set ............... 38
   5.1.1 Description ............................................. 38
   5.1.2 Initial Results ......................................... 39
   5.1.3 Variation with a Hybrid Structure for the Read Set .......... 44
   5.1.4 Results in other Scenarios ................................ 46
      5.1.4.1 1% Write Ratio .................................... 46
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>A transactional object in DSTM</td>
</tr>
<tr>
<td>4.1</td>
<td>Example of a versioned box</td>
</tr>
<tr>
<td>4.2</td>
<td>Example of a transaction’s read set</td>
</tr>
<tr>
<td>4.3</td>
<td>Example of a write set during execution</td>
</tr>
<tr>
<td>4.4</td>
<td>Example of a write set after execution</td>
</tr>
<tr>
<td>4.5</td>
<td>Example of a WriteSet divided in blocks</td>
</tr>
<tr>
<td>4.6</td>
<td>Example of the ATR in the original version of the JVSTM</td>
</tr>
<tr>
<td>4.7</td>
<td>Example of the ATR in the lock-free version of the JVSTM</td>
</tr>
<tr>
<td>5.1</td>
<td>Running time</td>
</tr>
<tr>
<td>5.2</td>
<td>Number of aborted transactions</td>
</tr>
<tr>
<td>5.3</td>
<td>Average execution time of a committed transaction (in microseconds; 96 threads, 250.000 transactions, 1.000 operations/transaction, 10% write ratio)</td>
</tr>
<tr>
<td>5.4</td>
<td>Running time in the original version, IdentityHashSet and hybrid read set (in seconds; 250.000 transactions, 1.000 operations/transaction, 10% write ratio)</td>
</tr>
<tr>
<td>5.5</td>
<td>Number of aborted transactions in the original version, IdentityHashSet and hybrid read set (250.000 transactions, 1.000 operations/transaction, 10% write ratio)</td>
</tr>
<tr>
<td>5.6</td>
<td>Performance of the original version, IdentityHashSet and hybrid read set (250.000 transactions, 1.000 operations/transaction, 1% write ratio)</td>
</tr>
<tr>
<td>5.7</td>
<td>Abort distribution between original version, IdentityHashSet and hybrid read set (250.000 transactions, 1.000 operations/transaction, 1% write ratio, 96 threads)</td>
</tr>
<tr>
<td>5.8</td>
<td>Average performance of a transaction (in microseconds; 250.000 transactions, 1.000 operations/transaction, 1% write ratio, 96 threads)</td>
</tr>
<tr>
<td>5.9</td>
<td>Performance comparison between original version, IdentityHashSet and hybrid read set (250.000 transactions, 1.000 operations/transaction, 50% write ratio)</td>
</tr>
<tr>
<td>5.10</td>
<td>Abort distribution between original version, IdentityHashSet and hybrid read set (250.000 transactions, 1.000 operations/transaction, 50% write ratio, 96 threads)</td>
</tr>
<tr>
<td>Section</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>5.11</td>
<td>Performance of the original version, IdentityHashSet and hybrid read set</td>
</tr>
<tr>
<td></td>
<td>(250,000 transactions, 1,000 operations/transaction, 100% write ratio)</td>
</tr>
<tr>
<td>5.12</td>
<td>Representation of the Active Transactions Record just before X validates</td>
</tr>
<tr>
<td>5.13</td>
<td>Running time</td>
</tr>
<tr>
<td>5.14</td>
<td>Number of aborted transactions</td>
</tr>
<tr>
<td>5.15</td>
<td>Running time</td>
</tr>
<tr>
<td>5.16</td>
<td>Number of aborted transactions</td>
</tr>
<tr>
<td>5.17</td>
<td>Performance of the adaptive IHS with adaptive validation with 1% write ratio</td>
</tr>
<tr>
<td>5.18</td>
<td>Performance of the adaptive IHS with adaptive validation with 50% write ratio</td>
</tr>
<tr>
<td>5.19</td>
<td>Example of an ATR under collaborative validation</td>
</tr>
<tr>
<td>5.20</td>
<td>Read set of transaction $T_6$ divided into blocks</td>
</tr>
<tr>
<td>5.21</td>
<td>Example commit queue</td>
</tr>
<tr>
<td>5.22</td>
<td>Example commit queue</td>
</tr>
</tbody>
</table>
List of Tables

3.1 Comparison between STMs .................................................. 21

5.1 Total running time (in milliseconds), number of aborted transactions and number of transactions committed/aborted per millisecond in the original version of the JVSTM and in the version with IdentityHashSet (250,000 tks, 1,000 ops/tx, 10% write ratio) .................................................. 39

5.2 Number of aborted transactions in each stage in the original version of the JVSTM and in the version with IdentityHashSet (250,000 transactions, 1,000 operations/transaction, 10% write ratio) .................................................. 41

5.3 Number of transactions that perform incremental validation, average number of incremental validation steps performed by each transaction and average time per single validation (250,000 transactions, 1,000 operations/transaction, 10% write ratio) .................................................. 43

5.4 Running time and number of aborted transactions in the original version, IdentityHashSet and hybrid read set .................................................. 45

5.5 Total running time of adaptive IHS and adaptive IHS with adaptive validation (in milliseconds, 10% write ratio) .................................................. 55

5.6 Comparison of different helper limit strategies .................................. 61

5.7 Comparison of write-back to disk without help and with simultaneous write-back .................................. 63

5.8 Comparison of the two different commit orders using simultaneous write-back .................................. 65

5.9 Comparison of different tweaks to the write-back help algorithm .................................. 69

5.10 Comparison of the original JVSTM with and without upgrade (48 threads) .................................. 70

5.11 Comparison of the original JVSTM with and without upgrade (96 threads) .................................. 70

5.12 Comparison of optimized JVSTM with and without upgrade (48 threads, with disk persistence) .................................. 71

5.13 Comparison of optimized JVSTM with and without upgrade (96 threads, with disk persistence) .................................. 71
# List of Code Listings

5.1  Code for adaptive validation .................................................. 51
5.2  Original write-back help algorithm ........................................ 66
5.3  Write-back help algorithm with AtomicInteger status flags ............ 67
5.4  Write-back help algorithm with AtomicInteger status flags and global
     AtomicInteger flag .......................................................... 68
1 Introduction

1.1 Motivation

Although the availability of multiprocessor computers is nothing new in the server domain, it is only in recent years that multicore processors became commonplace across most computing devices. However, this hardware potential does not translate into real performance gains unless applications are specifically designed to take advantage of parallel computing. Traditionally, programmers have turned to locks as the favorite tool to introduce parallelism into their applications. However, fine-grained locks become extremely error-prone and complex to design in large applications (Herlihy, Luchangco, Moir, and III 2003). On the other hand, coarse-grained locks, while less complex and error-prone, have limited parallelism and thus poor performance (Dice, Shalev, and Shavit 2006; Herlihy, Luchangco, Moir, and III 2003).

While the lock-based approach was somehow tolerable when concurrent programming was somehow a niche feature, the increased popularity of multicore processors demands a more intuitive and higher-level method for programmers to write concurrent applications.

To this end, Software Transactional Memory (STM) brings the transactional model, already proved and tested for several decades in the database community, to concurrent programming (Shavit and Touitou 1997). In simple terms, when using an STM system, the programmer only needs to identify operations that run concurrently and that share the same memory space, and divide them into transactions. With this information on hand, the STM system monitors which shared memory locations are accessed by which transactions and determines the presence of conflicts. A conflict arises when two transactions try to access the same memory location in conflicting modes (read/write or write/write). Upon detecting a conflict, the STM handles it automatically by aborting one of the conflicting transactions, rolling back its effects (if any) and, depending on the STM, restarting it. By doing all this automatically and dynamically, STM systems simplify the task of handling conflicts in concurrent applications, giving programmers more time to focus on the application itself.

However, the convenience of STMs compared to traditional approaches does not come without trade-offs. Keeping track of different transactions, and the memory locations they access, introduces space and time overheads over lock-based approaches. As a result, an application using well-designed fine-grained locks will perform better than it would using STM (Herlihy and Koskinen 2008). To be a viable substitute for locks, STMs have to reduce the overhead they introduce as much as possible while maintaining the ease of use.
CHAPTER 1. INTRODUCTION

1.2 Context

JVSTM started off as a lock-based system (Cachopo and Rito-Silva 2006; Cachopo 2007), but over time it eventually was developed to become lock-free (Fernandes and Cachopo 2011), but throughout all versions it was based on the concept of versioned boxes, which is its most distinctive feature. Essentially, a versioned box is a container for the history of a transactional object — a memory location that is shared by several transactions and can be accessed concurrently. By keeping a history of values instead of only the most recent one, JVSTM allows concurrent transactions that started at different points in time not to enter into conflict with each other under certain circumstances, which will be detailed later on.

Essentially, in the JVSTM transactions keep logs of their read and write operations during execution. The read log is used to guarantee that the transaction observed a consistent view of the system, while the write log is used to store written values locally instead of writing them immediately to the shared memory space, which means that other transactions will not observe temporary values that may never become definite.

If a transaction finishes without conflicting with another, it is validated using the read log to guarantee that the values it observed during execution remain consistent, and if so the transaction is put in a commit queue, which defines the order in which transactions are written to memory.

If the transaction is deemed invalid, its logs are discarded and the user is informed so that he can decide how to proceed, which typically means that the transaction will be restarted.

To keep each box’s history ordered, transactions have to be committed one-at-a-time in the order defined by the commit queue. Because only one transaction can be committed at a time, but several transactions can be waiting in queue to be committed, those that cannot be committed yet help the first one in the queue, thus speeding up the commit process.

1.3 Goals

The aim of this work is to analyze current STM systems and propose improvements to the scalability of one such system, the Java Versioned Software Transactional Memory (JVSTM). This system was chosen as the basis for this work due to the support of its authors and contributors.

The performance of an STM can be improved in two ways. The first one involves analyzing the different time and space costs associated to the execution of a transaction, and then changing the algorithms and data structures used by the STM engine in a way that reduces as much as possible the overhead imposed by the STM.

Another improvement path consists in using data structures that allow increased transaction concurrency, thus reducing the probability of a transaction aborting, which results in less aborted transactions and ultimately less time used to execute the same amount of work.

An example of this approach is Transactional Boosting (Herlihy and Koskinen 2008), which works by leveraging semantical properties of objects that STMs traditionally abstract themselves from. When employing this technique, programmers must specify inverse operations for each method that changes the state of an object.

This work follows the former approach, because it is more transparent to the programmer using the JVSTM in the sense that it requires no more intervention than the current implementation, whereas the latter approach requires that the programmer produces extra code to allow
increased concurrency, which may even not be possible in some application domains.

1.4 Outline

Chapter 2 starts by introducing a few basic concepts and key design alternatives of STMs. Then, Chapter 3 describes a few existent systems with different approaches and outlines their design choices in an attempt to understand why those choices were made and in what way they are expected to impact the system.

Then, Chapter 4 describes the JVSTM, the system whose scalability this work aims to increase, in detail to identify possible areas of improvement.

Chapter 5 presents several ideas on how JVSTM’s structures and algorithms could be changed to address potential problems, and also describes the results of those proposals.

Finally, Chapter 6 briefly sums up the key conclusions of this work and identifies areas that could be further developed in the future.


## 2.1 Basic Concepts of STMs

### 2.1.1 Transactions

The notion of transaction when applied to software translates into a set of operations, specified by the programmer, that is to be executed atomically.

In databases, transactions have to respect the following properties (Gray and Reuter 1993):

- **Atomicity**: either all actions of a transaction take effect or none of them does.
- **Consistency**: transactions shall take the system from a consistent state to another consistent state.
- **Isolation**: two transactions may share data between them, but they have to appear to run in isolation to an outside observer.
- **Durability**: after a transaction becomes effective, its results cannot be reverted and all transactions starting thereafter have to observe its results.

In software transactions, usually only the first three properties are considered (Moss 2006). Durability implies that the results of a successfully committed transaction should be stored persistently, which is not crucial to all application domains, and as such several STMs ignore this property.

After a transaction starts, it performs a sequence of write and/or read operations on shared memory objects and may terminate in two ways. If it executes successfully without entering into conflicts with other transactions, we say the transaction **commits**, i.e. its operations become effective and its results become accessible to new transactions. However, if during the execution or at commit-time conflicts are detected, the transaction **aborts** and is rolled back, which means it does not produce any changes to the system and must be restarted.

From atomicity and isolation follows linearizability, which states that transactions must appear to execute at one point in time between their start and their end, and that the order in which non-concurrent transactions appear to execute must be the same in which they started (Herlihy and Wing 1990).

**Opacity** (Guerraoui and Kapalka 2008) is another property of transactional memory systems that has the following requirements:

1. All operations of all committed transactions must appear to execute at one point in the respective transaction’s lifetime.
2. Operations executed by an aborted transaction cannot be observed by other transactions.

3. All transactions must observe a consistent view of the system.

Opacity ensures linearizability, but goes one step further by also requiring the consistency of transactions in all intermediate states, i.e. transactions must observe a consistent view of the system even while they are running.

Transactions are classified as read-only when they only read memory locations throughout their entire lifecycle, whereas a read transaction is one that reads at least one memory location at some point, independently of whether it performs write operations or not. The same logic applies for write-only and write transactions.

### 2.1.2 Transactional Memory

Also referred to as shared memory, this term refers to the part of the address space of an application that is accessed concurrently by transactions. These are the accesses that the STM engine has to monitor and synchronize in a way that minimizes conflicts as much as possible.

### 2.1.3 Transaction Nesting

The modular nature of most applications, where methods call other methods in cascade, means that applications using STM will have transactions invoking other transactions (subtransactions) at any point in their execution. Transaction nesting can have flat, closed or open semantics (Moss 2006):

- **In flat nesting**, subtransactions are subjugated to the invoking transaction. Each transaction can only have one subtransaction at a time and the parent does not run simultaneously when a subtransaction is active. Subtransactions operate directly on their parent’s read and write sets, and thus their effects only become globally visible when the main transaction successfully commits. In the event that a subtransaction aborts, the whole main transaction also aborts.

- **Closed nesting** has the same visibility property as flat nesting with regard to global memory locations, but in this case the effects of a subtransaction are visible to the main transaction when the former commits. Aborting a subtransaction only rolls back its changes and allows the main transaction to continue running, either by restarting the child transaction or by following a different course. Sibling transactions (child transactions that have the same parent) can run concurrently, but the parent transaction does not run while any of its children are active.

- Finally, in open nesting semantics a subtransaction’s results are not considered part of its parent, i.e. its read and write sets are not added to its parent’s sets. Instead, when a subtransaction commits its effects become globally visible.

The terms parent transaction and child transaction are used to denote a dependency between two transactions where the latter is invoked by the former. A top-level transaction is one that was not invoked by any other transaction, i.e. it does not have a parent.
2.1.4 Read and Write Sets

These sets are usually associated to each transaction and help the STM engine keep track of the transactional data read and written by each transaction, which is useful to detect conflicts between transactions. *Read sets* keep track of each read memory location — either the value itself or a version number —, while *write sets*, also called *redo/undo logs* as described below, provide a way for transactions to defer the update of transactional objects to commit-time.

2.1.5 In-place/Out-of-place Updates

A transaction that wishes to write some value in shared memory may do so in two different ways depending on how the STM engine synchronizes accesses to transactional memory:

1. We refer to *in-place updates* when a transaction writes tentative values to the shared memory location immediately during its execution, which implies that a lock must be acquired immediately and kept until the transaction commits.

2. *Out-of-place updates*, on the other hand, are not done in shared memory while the transaction is executing. Instead, the transaction stores tentative values in its local write set and only writes them to shared memory on commit.

2.1.6 Contention Management

Depending on the progress guarantees given by the chosen synchronization technique (as explained later in Section 2.2), the STM engine may need an external mechanism to solve conflicts between transactions and guarantee system progress in the event of livelocks or deadlocks. Such mechanisms usually carry the name of *contention managers*. They are contacted by transactions that detect a conflict and solve it by using a set of rules to decide how the conflicting transactions shall proceed to overcome the conflict. Such decisions may include aborting and retrying one of the conflicting transactions or making it wait (Herlihy, Luchangco, Moir, and III 2003).

2.1.7 Long-running Transactions

A transaction is said to be *long-running* when it executes a large number (typically millions) of actions, thus taking longer to finish and requiring access to more shared objects than smaller transactions (Cachopo and Rito-Silva 2006). Because of this, long-running transactions are more prone to enter into conflicts with other transactions. Also, because they can access a very large pool of transactional objects during their lifetime, their read and write sets will be larger. Depending on the algorithms and structures used, operations done on these sets, such as lookups, will potentially take longer and thus contribute negatively to the STM’s performance.

2.2 Design Alternatives

STM has been researched as an alternative to locks for more than a decade (Shavit and Touitou 1997), and as such several different proposals exist (Herlihy, Luchangco, Moir, and III 2003;
Saha, Adl-Tabatabai, Hudson, Minh, and Hertzberg 2006; Dice, Shalev, and Shavit 2006; Cachopo 2007; Damien Imbs and Michel Raynal 2008). This section describes the most relevant features and design alternatives that differentiate STM systems from one another, presenting their corresponding strong points and drawbacks.

2.2.1 Synchronization Technique

The ultimate goal of STM systems is to allow several transactions to run simultaneously and access shared data concurrently. Because two transactions cannot access the same shared memory location simultaneously when that constitutes a conflict, the STM system has to somehow synchronize those accesses. This synchronization can be either blocking or non-blocking (Saha, Adl-Tabatabai, Hudson, Minh, and Hertzberg 2006).

The blocking synchronization technique associates locks to shared data and presupposes that those locks are acquired by the transaction prior to data being effectively changed and released after the transaction commits or aborts. Also, these locks can be acquired either at encounter or at commit-time. Encounter-time lock acquisition means locks are acquired as soon as a transaction tries to access the data, while the commit-time strategy defers lock acquisition until the transaction tries to commit. Note that all these lock-related operations are always done automatically by the STM and not by the programmer. Problems derived from the usage of locks such as deadlock and starvation are usually avoided by contention managers.

STMs that rely on non-blocking synchronization do not operate directly on the shared data, but on copies of it instead. Each of these copies is local to a single transaction and thus can be manipulated through the whole lifecycle of the transaction without synchronization concerns. On commit, the shared data is atomically overwritten with the new values. Non-blocking synchronization can have progress guarantees with different strengths (Herlihy, Luchangco, and Moir 2003):

- **Obstruction-free** techniques “guarantee progress for any thread that eventually executes in isolation”, i.e. in the absence of conflicts with other threads attempting to access the same transactional objects. This is the weakest progress guarantee possible and requires a contention manager to guarantee system progress when two transactions conflict with each other.

- **Lock-free** synchronization is stronger than the precedent by guaranteeing that at least one thread always makes progress.

- The strongest progress guarantee possible in a synchronization technique is **wait-freedom**, as in this case progress is guaranteed for all threads, independently of contention.

Wait-freedom offers the ideal behavior, but techniques with weaker guarantees are often enough to make progress and allow for simpler implementations (Herlihy, Luchangco, and Moir 2003).

The concepts of in-place and out-of-place memory updates introduced in the previous section are closely related to the choice of synchronization technique. If an STM wants to update memory locations in-place, it must necessarily use locks to gain immediate exclusive access to the memory location it wants to update. Otherwise, a transaction could update some memory...
location in-place even while another transaction was accessing the same memory location. In the case of out-of-place updates, transactions operate on copies of the data and not the shared data itself, and thus the use of locks for synchronization is not required.

### 2.2.2 Transactional Data Granularity

Because conflicts between transactions arise from concurrent access to data, its granularity has an impact on the number of conflicts.

**Word-based** STM engines detect conflicts based on data accesses the size of a memory word or cache line. Because of this fine granularity, conflicts between transactions are minimized, since two transactions would have to be accessing the exact same field in order to conflict with each other. On the downside, such granularity generates increased time overhead because the STM has to intervene each time the application wants to access a word, which is likely to happen very often. It also requires more space to be used because the read set will also contain more entries. This becomes particularly obvious when applied to object-oriented applications. Considering a simple class with five integer attributes, a word-based STM would require five separate STM API calls if the application tried to access all five fields of an object of that class, whereas an **object-based** STM would require a single call.

However, the overhead advantage of object-based STMs comes at the expense of unnecessary conflicts that arise from two transactions trying to access different fields of the same transactional object. Considering the same class from the previous example, a word-based STM would allow up to five concurrent transactions to access the same object (provided each was accessing a different attribute), whereas an object-based STM would only allow one transaction at a time, independently of the attributes being accessed.

### 2.2.3 Concurrency Control

As previously mentioned, blocking STMs synchronize shared data accesses by means of locks. These can be of two types: **read/write locks** or **versioned write locks**.

The **read/write locking** method allows transactions to acquire different locks for reading and for writing, depending on what operations they will perform on the shared data. This locking mechanism allows several transactions to read the same object concurrently, but write operations are incompatible with any other data access. This mechanism has the advantage of being a very common concurrency control choice, and thus there are several implementations of it available for different programming languages and platforms. However, its performance is poor because (1) it requires locks to be acquired for read operations and (2) acquiring read or write locks that are already taken makes transactions wait until the locks are available.

**Versioned write locks** do away with locks for read-only operations by keeping track of a version number for each shared object, which allows read operations to be invisible to other transactions. Every time a transaction reads a shared object, its version is kept in the transaction’s read set. At commit-time, the transaction verifies if the values it read are up to date by comparing the version numbers in its local read set with the most current version numbers.

Like in read/write locking, write operations are synchronized by a lock, but read operations only need to check if the lock is free and do not need to actually acquire it, which yields better performance (Saha, Adl-Tabatabai, Hudson, Minh, and Hertzberg 2006).
2.2.4 Lock Acquisition and Recovery Strategy

As a transaction may either commit or abort, the STM engine must keep track of the changes made to the shared data until the atomicity of the transaction can be guaranteed.

The undo log strategy works by allowing transactions to do in-place updates of the shared data and keeping a log of the old values. When the transaction successfully commits, the STM engine only needs to discard the undo log. In the event of an abortion, the STM replaces the shared data values with the previous versions stored in the undo log. This requires immediate (encounter-time) locking of the shared data in order for the in-place update to be possible and to prevent other transactions from reading the still tentative value.

In the redo log strategy transactions do out-of-place updates of the memory locations by storing them in a write set. When the transaction commits, the shared data is overwritten with the new values stored in the transaction write set. However, if the transaction fails, the abort operation consists only in ignoring the redo log because no shared data was actually changed. This strategy allows the STM engine to defer lock acquisition to commit-time.

When a transaction wants to read a memory location in the redo log strategy, it must first search for it in its write set before searching in the shared data pool, because it might have written in that same memory location before. This additional search represents an overhead over the undo log strategy, which may impact the performance of read operations. However, the undo log method has the disadvantage of holding on to locks for a long period of time, while redo log only holds them during the commit phase, which is of special importance in long-running transactions.

2.2.5 Lock Placement

Blocking STM engines have two options regarding where to store the locks associated to each transactional object, namely next to the object itself (inline) or in a separate table.

Storing locks adjacent to the data has the great advantage that both the object and the lock will probably be present in the cache at the same time (Saha, Adl-Tabatabai, Hudson, Minh, and Hertzberg 2006), yielding less cache misses and thus having greater performance. However, the developer cannot use previously defined data structures without changing their definition and how they are handled, or otherwise they would not be lock-aware.

On the other hand, if locks are kept in a separate table developers need not change existing libraries. Locks are stored in the table by associating a memory location to a table position by means of a hash function, which is prone to collisions and thus may impair performance.

2.2.6 Transaction Validation

Before the results of a transaction become definite, i.e. before it can be successfully committed, the STM engine must first validate it. Essentially, validating a transaction consists in checking whether committing it would go against any of the properties an STM system is expected to preserve, i.e. if it conflicts with any other transaction (Herlihy, Luchangco, Moir, and III 2003). This is done by observing which shared data locations a given transaction has accessed for reading and/or writing and it can be done at two different times:
2.2. DESIGN ALTERNATIVES

- **Pessimistic** STMs assume that a transaction can enter a conflict at any time and thus check for conflicts each time the transaction tries to access the shared memory space. Each validity check results in the abortion of the corresponding transaction in the event of a conflict or allows it to continue otherwise, thus preventing transactions from running any further unnecessarily.

- **Optimistic** STMs assume all transactions will be successful and thus allow them to run speculatively and defer all validity checks to commit-time. Such validity checks are required to guarantee a transaction’s linearizability, which is done by checking if all values read by the transaction, i.e. its read set, are still up to date and have not been overwritten by any other transaction. Following this validation approach may waste CPU time by allowing transactions to continue running even when there is already enough information to know they should be aborted, but on the other hand continuous (and potentially costly) validity checks are avoided.
This chapter describes some existing STM systems, detailing their protocol, what features they implement, what design options they make and, where applicable, new approaches they apply to increase scalability that differentiate them from the rest. Some systems were chosen because they are common references in related literature, and others because they offer a different combination of design choices.

### 3.1 DSTM

*Dynamic Software Transactional Memory* (DSTM) (Herlihy, Luchangco, Moir, and III 2003) relies on non-blocking synchronization and has a pessimistic approach to validation. It introduces modular contention managers as the way to guarantee obstruction-free thread progress.

Transactional data consists in special container objects (see Figure 3.1) that encapsulate normal objects. When the programmer wants to create a transactional object, he does so by first instantiating an object in the normal way and then by assigning it to a new container object. From this moment on, the programmer must be careful to only access this transactional object using DSTM’s transactional API, or else DSTM will not be able to monitor object accesses properly, and thus will behave erratically. This mechanism presupposes that each encapsulated class offers a cloning method, which is used by DSTM to create a different working copy of the object for each transaction that tries to access it. Each copy is assigned a version number based on the accessing transaction’s identification. It is because of this that the engine does not need locks, but on the other hand requires classes to implement a special interface and makes reusing existing code less practical.

![Figure 3.1: A transactional object in DSTM](Image)

Each container object points to a Locator object, which has a reference to the last transaction that opened the object for writing, as well as references to new and old versions of the object. The status of a transaction can be active, committed or aborted. When a transaction wishes to read or write into a transactional object, DSTM checks the status of the transaction referenced by the Locator associated to the object:
• If the transaction is marked as active, the Old Object field in the Locator is referencing the most up to date committed version and New Object is a tentative, uncommitted version. Thus, Old Object is returned to the transaction opening the shared object.

• If the transaction referenced by the Locator is marked as committed, New Object is the most recently committed version.

• If the status is aborted, New Object contains a discarded version and Old Object is the value to be used.

If the object is being accessed for reading, the appropriate value is stored in the transaction’s read-only table, a kind of read set. If the access is for writing, DSTM creates a new Locator object with the following values:

1. Transaction references the transaction that is now writing into the object.

2. New Object references the new working copy of the object created using the clone method.

3. Old Object references the old committed version as determined by the rules described above. If the transaction that last wrote into the memory location is still active, a conflict arises and the contention manager has to be called to decide how both transactions will proceed.

Finally, the system changes the container’s reference to the newly created Locator object, the old one is discarded and the transaction that asked to access the memory location writes into the New Object field. Changing the Locator reference is done atomically using a CAS instruction, which guarantees that only one transaction overwrites the Locator object.

DSTM takes a pessimistic approach to transaction validation to detect concurrent accesses to shared data and abort conflicting transactions. To validate a transaction, DSTM checks whether any memory location it accessed has been opened afterwards in a conflicting mode by another transaction. This is done to detect inconsistencies that could arise if during the execution of a transaction $T_1$ another transaction $T_2$ changes some shared locations that $T_1$ accessed before and others that it will access thereafter. Without this validation, $T_1$ would observe an inconsistent snapshot of the application’s shared memory, only observing a portion of $T_2$’s effects.

**Commit and Abort protocols** Committing or aborting a transaction is as simple as validating its read-only table and, depending on this validation’s result, atomically changing the Status field to the appropriate value. From that point on, any transaction that tries to access an object updated by this transaction will be pointed by the engine to the most recent version determined by this transaction’s status.

Due to the use of a CAS instruction to swap Locator objects during transaction execution, DSTM’s memory location update policy can be considered to be in-place. However, in the event that a transaction $T$ aborts, DSTM does not need a separate undo log to reinstate old values, as these are kept in each Locator’s OldObject field, which other transactions will read if $T$ is marked as aborted.

Of particular interest is the concept of early release that DSTM’s authors introduce. Fundamentally, it allows the programmer to release during execution an object that was previously
opened for reading in a transaction. By doing so, this object will not conflict with any other
transactions that might need it, but it is up to the programmer to determine when it is safe to
release objects in this way, otherwise it may lead to inconsistent states. If done successfully,
performance gains will arise from reduced conflicts, even in long-running transactions, but the
burden on the programmer is arguably similar to that of designing fine grained locks (Saha,
Adl-Tabatabai, Hudson, Minh, and Hertzberg 2006).

3.2 McRT-STM

Unlike most STMs, which are standalone, McRT-STM (Saha, Adl-Tabatabai, Hudson, Minh,
and Hertzberg 2006) is a system developed in the context of an experimental multi-core runtime
(McRT) that also includes a thread scheduler, a synchronization framework and a memory
manager, components which the STM leverages to increase performance.

McRT-STM’s authors argue that other STM implementations guarantee non-blocking prop-
erties at the expense of simplicity and performance. Therefore, McRT-STM is a blocking
STM where non-blocking properties are enforced externally by the runtime’s thread scheduler
through preemption control, with minimal intervention from the STM system.

McRT-STM associates a versioned write lock with each transactional memory location,
with both object and word-based conflict detection methods being available to the programmer.
The choice of implementing both methods is sustained by inconclusive performance results in
the comparison between them using different data structures. When a transaction wishes to
update a shared memory location and does not own the associated lock already, it acquires
said lock if it is free or invokes the contention manager, which determines if the transaction
should wait until the lock is released or abort. Once the lock is successfully acquired, it is
added to the transaction’s write set and the data is then updated in-place. On the other hand,
read operations only need to check if the lock associated to the data is not held by another
transaction and do not actually acquire it. Depending on which transaction owns the lock, the
read operation proceeds as follows:

- If the lock is free, the transaction reads the value and stores the corresponding version
  number in the read set.
- If the lock is held by the same transaction that wants to read the location, access is
  granted immediately.
- If the lock is held by another transaction, the memory location cannot be read and the
  contention manager is called to determine if the transaction should wait and retry or
  abort.

The versioned write lock approach eliminates writing to the lock word in read operations and
upgrading locks from read to write access, which the authors point out as performance degrad-
ing factors. This STM has a semi-pessimistic approach to transaction validation by inserting
validation checks at backward edges to avoid error conditions such as infinite loops.

Commit Protocol The commit operation does not need to install any values because up-
dates were already done in-place. However, it must validate the transaction because read op-
erations did not require locks to be acquired, which means values might have been updated by
other transactions that committed during the execution of this transaction. The transaction’s read set is validated by comparing the version numbers it contains with the most current versions of the data. After this is done successfully, the locks can be released and the transaction marked as committed.

Abort Protocol Because of the in-place data updates, aborting a transaction requires the old values to be restored from the undo log to memory, after which all locks held by the transaction can be released and the contention manager invoked to restart the transaction.

3.3 TL2

Transactional Locking II (TL2) (Dice, Shalev, and Shavit 2006) is a lock-based approach that uses a global version-clock to timestamp transactions. One of the objectives of this work is to improve on previous lock-based systems that rely on specialized managed runtime environments, such as that of McRT-STM.

Like McRT-STM, TL2 associates a versioned write lock with each memory location, but in this case the lock’s version number is not local to each object and is taken from the global version-clock instead. TL2 also supports both word and object-based approaches as well as adjacent or separate lock placement. Transactions are executed speculatively, which means that memory locations are updated out-of-place and locks acquired at commit-time.

Write transactions start by storing a sample of the global version-clock locally, which will be used to detect changes in data. While the transaction is running, read and write sets are kept for future validation and installment of new values, respectively. The read set only stores the addresses of memory locations read by the transaction, while the write set also stores the speculative value that the transaction wishes to write. Read operations are augmented with a validation check that verifies if the lock associated with the data to be read is free (but does not acquire it) and asserts that its version number is at most equal to the value of the global version-clock previously sampled by the current transaction. Also, the lock’s version number must be the same immediately before and after the read operation. This is done to guarantee that locations read by the transaction have not been changed since its start by other transactions.

With performance in mind, read transactions have a simpler protocol than writing ones where no write set is kept. Therefore, their execution consists only in validating each read operation the same way as write transactions do. TL2 supports two methods of differentiating between read and write transactions:

1. Developers can mark transactions as read-only at compile time.
2. The STM engine assumes at first that all transactions are read-only. If a transaction attempts a write operation, it is aborted and marked as a write transaction, so that future transactions that execute the same code already begin in write mode.

Commit protocol Locks for each location in the write set are acquired at commit-time in such an order that avoids deadlock. The commit operation then proceeds by incrementing the global version-clock and revalidating the read set, which is done again by comparing each location’s current version to the transaction’s global version-clock sample. Despite the read set
having been validated during execution, revalidation of memory locations read by the transaction is necessary because other transactions could have updated them since they were read. After the read set is revalidated, the values stored in the transaction’s write set are installed (redo log strategy), each lock’s version is updated to the new global version-clock’s value and all locks are released, after which the transaction is successfully committed.

**Abort protocol** The abort protocol consists in dropping the transaction’s write set, if applicable, and releasing any locks it may hold. Transactions can abort at three different times:

1. When the read set cannot be validated, either during execution of a transaction or at commit-time.
2. During acquisition of locks for the write set at commit-time, if any of them is taken.
3. When a transaction was assumed to be read-only but attempts to make a write operation.

A few of TL2’s approaches to increase performance include acquiring locks only at commit-time, tweaking the version-clock implementation for low contention and adopting Bloom filters (Bloom 1970) to determine while transactions are running whether a memory location is present in their write set.

As a way to reduce contention generated by the global version-clock when transactions attempt to increment it, the authors propose augmenting the clock variable, which includes a version number, with the id of the thread that last updated it. By doing this, “a thread will not need to change the version number if it is different than the version number it used when it last wrote”, which reduces the number of times the version-clock variable is updated and thus it generates less contention.

When a writing transaction wants to perform a read operation, it must first search for the value in its write set. This is necessary because the same transaction may have previously written a new value in the memory location that is now being read, in which case the read operation should return the most up to date tentative value from the write set instead of the value stored in memory. TL2 resorts to the aforementioned Bloom filters to perform this search more efficiently. Bloom filters are an ideal data structure for this because their strength is identifying a given message as not being a member of a given set, and they do it with higher time and space efficiency than traditional hash-coding methods. By ruling out candidate memory locations using Bloom filters, the number of costly and unnecessary searches in the write set is greatly reduced. However, Bloom filters allow false positives, which means that a small fraction of unnecessary searches in the write set will still be performed.

### 3.3.1 CS-STM

Despite not being a completely new STM, I chose to include Consistent State Software Transactional Memory (CS-STM) (da Cunha 2007) in this analysis because it is a work similar to the one this thesis proposes to do, in the sense that it takes a state-of-the-art STM (TL2) and changes its implementation with the goal of observing the impact of such changes. CS-STM introduces the following new features:

---

1In this context, a message corresponds to a memory location.
• **User-called aborts** that allow the programmer to explicitly abort a transaction on demand. This is useful because programmers can take advantage of the associated recovery mechanisms in specific conditions that depend on the application’s semantics and not only on the STM’s transactional logic.

• TL2 only guarantees that transactions commit successfully *at most once*, whereas CS-STM provides **automatic transaction retries** that guarantee that each transaction is committed successfully *exactly once*.

• **Transaction nesting** was also added to TL2 with an implementation partially inspired in closed nesting semantics. The behavior in the event of an abort is different depending on whether the transaction is aborted explicitly by the programmer or by the STM due to a conflict.

Other changes were made to TL2 with performance in mind. The first of such changes is the introduction of an undo log mode with in-place updates and encounter-time lock acquisition, unlike TL2’s redo log strategy with out-of-place updates and commit-time lock acquisition. Like in TL2, read operations are augmented with consistency validations, but CS-STM has three alternative levels of validation (described below) that provide different guarantees and performance.

**Commit/Abort protocol** Write operations in CS-STM consist in acquiring the lock and installing the new value immediately instead of storing it in a redo log. Old values are stored in an undo log to be used if the transaction aborts. This approach reduces the time needed to commit a transaction because the new values are already in memory. Using an undo log comes at the cost of increased time in abort operations, which have to reload old values from the log back into memory, but it should not be a problem as aborts ought to be far less frequent than successful commits.

Another improvement proposed by CS-STM is to change the block size when working in word-based mode. With increased block size, the STM needs to have less locks and thus generates less overhead. However, the greater the block size, the more unnecessary contention will be generated.

The author describes three consistent state validation modes with different requirements and guarantees that can be used alongside object-sized data granularity:

• **Full state validation** requires the lock’s version to be verified every time a memory location is read, thus guaranteeing the transaction’s consistency at all times but generating more overhead. This can be done automatically by the STM engine.

• **Partial state validation** places validation checks only in some locations. These locations are chosen by the compiler or by a specialized optimization tool based on sequential accesses to the same memory location. If a given memory location is accessed by several sequential instructions within a transaction, its consistency will only be verified once after the last instruction and not every time after each instruction. Inconsistencies are allowed but will only last while the same memory location is being accessed repeatedly.

• **No state validation** is an optimistic approach that does not verify consistency while the transaction is running.
3.4 A LOCK-BASED PROTOCOL FOR STM (LBP-STM)

Performance-wise, CS-STM’s tests show small advantages of the undo log strategy and adjacent lock placement over their respective alternatives. Results also show a more pronounced advantage of object-based locking over the word-based approach, but the difference becomes small when the number of conflicts rises. As expected, partial state validation clearly shows better results than the full state mode and no state validation overtakes both approaches in most tests, even if not by a great margin.

3.4 A Lock-based Protocol for STM (LBP-STM)

As implied by its name, A Lock-based Protocol for STM (LBP-STM) (Damien Imbs and Michel Raynal 2008) describes a blocking protocol that uses reader/writer locks to synchronize concurrent accesses to transactional memory locations. These locks are acquired at commit-time and memory locations are updated out-of-place.

Apart from the lock that all other STM systems described in this section associate with transactional memory locations, LBP-STM also associates the following two sets to each shared object $O$:

1. A read set $RS$ that keeps track of which transactions read the object, unlike conventional STMs that only keep a read set associated to each transaction.

2. A forbidden set $FBD$ that keeps track of transactions that have read another object that has since been updated by another transaction that has already committed. A transaction belonging to this set should not read $O$ again from shared memory, since it conflicts with a transaction that has already committed. This allows the conflicting transaction to be aborted as soon as the conflict arises.

This system also keeps a global set $OW$ of all transactions that have read values that were later updated by other transactions, which is used to detect conflicts between transactions.

Additionally, each transaction also keeps read and write sets, which are queried using Bloom filters for efficiency. Unlike traditional write sets, these only store the ids of the memory locations written and not the actual values.

When a transaction $T$ wishes to update a shared memory location $O$, it starts by allocating space for a local copy of $O$ if it does not already exist. $T$ then proceeds to write the new value in the local version of the memory location and adds its id to $T$’s write set. All of these operations work exclusively in $T$’s local memory and thus never cause it to abort.

When a local working copy of the memory location to be read ($O$) is already available because it was previously accessed by the same transaction, the read operation consists only in returning the value stored locally. However, if there is no local working copy of $O$ in transaction $T$, the protocol for read operations is the following:

1. Space for a local working copy of $O$ is allocated.

2. $O$ is added to $T$’s read set.

3. $T$ acquires a lock to the shared object $O$. 
4. The current value of \( O \) is copied from shared into local memory.

5. \( T \) is added to \( O \)'s read set (\( RS \) as previously explained).

6. The lock is released.

7. If \( T \) is present in \( O \)'s forbidden set \( FBD \), \( T \) is aborted.

8. The value is returned from local memory.

Step 7 is necessary because another transaction could overwrite \( O \)'s value between the lock being released and the value being returned to the transaction, thus invalidating \( T \)'s local working copy of \( O \).

**Commit protocol** Read-only transactions are committed immediately because all read operations already triggered validation checks. Write transactions follow a more complex protocol:

1. Locks for all objects in the transaction’s read and write set are acquired.

2. If the committing transaction belongs to the global set \( OW \) described above, it aborts. Otherwise, it is guaranteed to commit successfully.

3. The new values for the memory locations in the write set are copied from the transaction’s local memory into shared memory.

4. The global set \( OW \) is updated to include all transactions that read the memory locations updated by \( T \).

5. The sets \( FBD \) associated to all objects updated by the committing transaction \( T \) are also updated to include the transactions that read overwritten values.

6. The locks for all accessed memory locations are released.

**Abort protocol** As described above, a transaction can only abort either while reading a value or during commit, not while writing into a memory location. Also, aborts triggered during the commit operation are always associated to invalid read operations. From this follows that write-only transactions are never aborted.

### 3.5 Summary and Conclusions

Table 3.1 presents a comparison between all STM systems described in this section using some of the most relevant design alternatives presented in Section 2.2. Note that CS-STM, being based on TL2, also has a redo log mode with commit-time lock acquisition and out-of-place memory updates that is not represented in the table. Also, LBP-STM’s lock placement policy is missing because the authors did not specify it.

As expected, blocking STMs that update memory in-place require encounter-time lock acquisition, whereas out-of-place updates only need locks to be acquired at commit-time.

Also of note is that no system makes a clear choice between placing locks adjacent to the data or in a separate table. While CS-STM shows an advantage of adjacent locks in most tests, it is too small for it to emerge as a clear winner in terms of performance.
Table 3.1: Comparison between STMs

<table>
<thead>
<tr>
<th></th>
<th>DSTM</th>
<th>McRT</th>
<th>TL2</th>
<th>CS</th>
<th>LBP</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Synchroniz. technique</strong></td>
<td>Non-blocking</td>
<td></td>
<td></td>
<td></td>
<td>Blocking</td>
</tr>
<tr>
<td><strong>Data granularity</strong></td>
<td>Object-based</td>
<td>Word or object-based</td>
<td></td>
<td>Object-based</td>
<td></td>
</tr>
<tr>
<td><strong>Concurrency control</strong></td>
<td>—</td>
<td>Versioned write locks</td>
<td></td>
<td>R/W locks</td>
<td></td>
</tr>
<tr>
<td><strong>Lock acquisition</strong></td>
<td>—</td>
<td>Encounter-time</td>
<td>Commit-time</td>
<td>Encounter-time</td>
<td>Commit-time</td>
</tr>
<tr>
<td><strong>Memory update</strong></td>
<td>In-place</td>
<td>In-place</td>
<td>Out-of-place</td>
<td>In-place</td>
<td>Out-of-place</td>
</tr>
<tr>
<td><strong>Lock placement</strong></td>
<td>—</td>
<td>Adjacent or separate</td>
<td></td>
<td>Not specified</td>
<td></td>
</tr>
<tr>
<td><strong>Validation</strong></td>
<td>Pessimistic</td>
<td>Pess. or Optim.</td>
<td>Pessimistic</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

3.5. SUMMARY AND CONCLUSIONS
CHAPTER 3. RELATED WORK
In this chapter I will describe the JVSTM more thoroughly, with a special focus on the features and data structures that are more relevant for the optimizations I will propose in Chapter 5.

JVSTM started off as a lock-based system (Cachopo and Rito-Silva 2006; Cachopo 2007), but over time it was subjected to several revisions, and eventually a lock-free version (Fernandes and Cachopo 2011) was developed, but the concept of versioned boxes is common to all versions. Unless specifically stated, the features and behaviors described henceforth refer to the lock-free version, as this is the version this work will try to improve upon. Moreover, there are other iterations of the lock-free JVSTM that use different data structures, but those will not be described here to avoid confusion.

Section 4.1 introduces the versioned memory model, which is JVSTM’s most distinctive feature. Then, Section 4.2 describes the phases of a transaction’s lifecycle and Section 4.3 focuses on the most important data structures used by JVSTM in general and in each transaction. Finally, Section 4.4 describes the benchmark used to assess JVSTM’s performance in several scenarios, as well as some conclusions that enabled the identification of possible optimizations.
4.1 The Versioned Memory Model

As one would expect from any STM, in JVSTM a transaction is a set of operations, marked by the user, that should execute atomically. Transactions can run at one of two different levels — a nested (or child) transaction is one that is started by another transaction (its parent), whereas a top-level transaction does not have a parent.

Each top-level transaction is executed by a thread and its child transactions (if it has any) are also executed by that same thread. From this it follows that a nested transaction and its parent never run simultaneously, and the same applies to two child transactions that share the same parent. This behavior is known as linear nesting (Cachopo 2007; Moss and Hosking 2005).

When a top-level transaction starts, it is assigned a temporary number that is the same as the one from the most recently committed transaction, which means that several concurrent transactions may have the same number. However, once a transaction is successfully validated and enters the commit queue, it is assigned a unique number in such a way that all transactions in the commit queue are numbered sequentially. A nested transaction is assigned the same number as its parent and upon commit it is not assigned a new number because its changes are committed to the parent transaction and it does not enter the commit queue.

The most distinctive feature of JVSTM is its multi-version memory model. Under this model, each shared memory location is represented by a versioned box, implemented by the VBox class described in Section 4.3.1. Each box encapsulates multiple versions of a shared memory location, i.e. it contains that location’s history. Each value is associated with a version number that corresponds to the final number of the transaction that committed that version. Like other STMs’ memory locations, versioned boxes support two operations — read and write. While reads happen during a transaction’s execution, writes to a box only happen at commit-time; tentative values are written in the transaction’s write set.

Having multiple versions of the same memory location allows JVSTM to guarantee that all read-only transactions always complete successfully, because even if another transaction commits a newer value to that box, the read-only transaction can still access the version it needs. This guarantee also eliminates the need for validation in this kind of transactions. Because read-only transactions always observe a consistent state that is defined by the number they were assigned at start, it follows that such transactions are linearizable at the time they started (Cachopo 2007).

On the other hand, write transactions can only be linearized at the time they commit, because that is when their changes become visible to the rest of the system. However, they ran with a consistent view of the system at the time they started, which means these transactions’ read sets have to be validated at commit time, as explained in Section 4.2, to ensure their validity at that point. Because write-only transactions have an empty read set, their validation is always successful and they can always be linearized at commit-time, which means that, like read-only transactions, they never abort.

4.2 Transaction Lifecycle

In JVSTM, a typical top-level transaction’s lifecycle can be divided into three main stages: begin, execution and commit/abort.
4.2. TRANSACTION LIFECYCLE

4.2.1 Begin

The begin phase is the shortest of all three and essentially consists in initializing fields and data structures that will be used in the subsequent phases. For example, it is in this phase that a transaction is assigned a temporary version number, the same as the most recently committed transaction’s number, that will define the context under which the transaction will run, i.e. which version of the transaction memory it will “see”.

4.2.2 Execution

The execution phase is when the transaction’s actual work is done by reading and/or writing boxes. In the case of write transactions (both write-only and read/write), read and write operations are logged in the transaction’s read and write sets, respectively. On the other hand, read-only transactions do not require a read set because they do not need to be validated, as explained in Section 4.1. The absence of a read set means read-only transactions have a lower overhead than write transactions, which was one of JVSTM’s original goals (Cachopo 2007). Write-only transactions are also guaranteed to commit successfully, but their write operations must be logged so that other (read/write) transactions can validate themselves against the write-only ones.

During the execution phase, transactions can abort under the following circumstances:

- When a read-only transaction attempts to perform a write operation on a box. This gives the user the possibility of assuming a transaction is read-only when it starts, thus taking advantage of the lower overhead of this kind of transaction, and only restart it as a read/write transaction (with higher overhead) if indeed that need arises.

- When a read/write transactions tries to read from a box that has been updated after the transaction started (i.e. the box’s most recent version is greater than the transaction’s, in which case we know the transaction cannot be committed).

4.2.3 Validation and Commit

This stage only applies to write transactions because read-only transactions do not need to be validated and cannot abort. For write transactions that finished the execution phase without conflicts, this third stage can be divided into three parts — write-back (first), validation and write-back (second), whereas those that encountered a conflict during execution skip this stage and abort.

Writing back a transaction consists in making its results permanent and visible to other transactions by inserting them in their corresponding boxes. The first write-back step ensures that the write sets of all transactions that are already validated and in queue to be committed are written-back. When the transaction cannot observe any more pending transactions in the queue, it performs a two-phase validation process. The commit queue, also called Active Transactions Record and explained in more detail in Section 4.3.4, is essentially a linked list that represents the order in which valid transactions are committed. To guarantee that the queue only contains transactions that can effectively be committed without
compromising the system’s consistency, each transaction must be validated before entering the commit queue.

The first step of validation, called snapshot validation, consists in validating the transaction’s whole read set by checking whether any box was updated in the meanwhile by another transaction. To do this, JVSTM checks if the most recent version of each box referenced in the read set still has a version number at most equal to the transaction’s number — if all boxes in the read set meet this criteria, the transaction can be considered valid, otherwise the transaction is aborted immediately. If snapshot validation succeeds, the transaction tries to put itself in the commit queue by performing a CAS operation. If the CAS succeeds, the transaction enters the queue and does not need further validation. If it fails, we know that another competing transaction succeeded in entering the queue, and thus the transaction where the CAS failed must revalidate itself. However, instead of validating the entire read set again, it performs incremental validation, which consists in looking for intersections between the transaction’s read set and the other transaction’s write set (the one where the CAS succeeded). This second type of validation is optional in the sense that it only happens when the CAS fails, whereas snapshot validation is always performed exactly once by all transactions. A single transaction may fail to perform the CAS more than once (if more than one transaction is competing with it) and it must perform one incremental validation for each transaction where the CAS succeeds. Essentially, the idea behind the two-phase validation process is to guarantee that a transaction is valid against all transactions that precede it in the queue — snapshot validation guarantees validity up to the point when the first attempt is made to enter the queue, and incremental validation handles any transactions that enter the queue afterwards.

Once the transaction has successfully validated itself and entered the commit queue, it again helps all pending transactions in the queue, up to and including itself, to write-back their write sets. Both instances (pre- and post-validation) of the write-back process are collaborative in the sense that several transactions can take part in writing back one transaction’s write set, which is possible due to the WriteSet structure described in the next section. In its initial lock-based version, JVSTM only allowed one transaction to be in the commit phase at any time, which meant that other transactions waiting to commit remained locked. This was necessary to guarantee that transactions’ write sets were written back in the order the transactions were validated without conflicts. By using collaborative write-back, the lock-free version still requires one transaction to be written back at a time, but also allows pending transactions to take part and speed up the process, instead of keeping them idle.

4.2.4 Abort

As described in this section, top-level transactions in JVSTM can abort either during execution or validation. In either case, no permanent changes were made to the shared boxes and the transaction was not put in the commit queue, which means that no actions need to be rolled back to keep the system consistent. Instead, either a WriteOnReadException or CommitException is thrown, depending on the reason, and the user can decide how to proceed.
4.3. **Important Data Structures**

4.3.1 **VBox**

As previously explained, JVSTM’s most distinctive feature is that each memory location is encapsulated in a box capable of holding several versions of a shared variable. Each box is implemented using the `VBox` class, which has three fields: `tempValue`, `currentOwner`, and `body`.

The first field has the purpose of allowing a running transaction, identified by the `currentOwner` field, to write a tentative value directly into the box instead of using the traditional write set. As described in Section 4.3.3.1, this speeds up certain scenarios while still guaranteeing that this temporary value is not observed by other transactions until it is committed permanently. Absent from the first versions of JVSTM, this feature simulates a property of in-place updates that other STMs have — when a transaction reads from a box where it previously wrote and that it owns (i.e. it is its `currentOwner` and it wrote the value in `tempValue`), it can read directly from `tempValue` instead of searching in the write set. Also, when writing again in that box, it can simply overwrite directly the `tempValue` without having to update the write set, whereas without this feature the write set would have to be updated each time. Unlike in-place writes though, in this case the temporary value is not accessed by other transactions and does not have to be reverted if the transaction aborts.

The last field, `body`, references a linked list of bodies, where each body holds one version of the memory location. This list only holds values that are permanent, i.e. those that stem from committed transactions, whereas tentative values are either stored in the `tempValue` field or the running transactions’ write sets. When a new version is committed by a transaction, a new body with the new value and version number is created and inserted into the start of the list using a CAS. From this follows that the list of bodies is ordered starting with the most recently committed version.

When a transaction $T_A$ wants to read directly from a box during execution, it cannot simply read the most recent version. Instead, $T_A$ must traverse the bodies list to look for a version compatible with its own transaction number in order to guarantee that it observes a consistent view of the system. Because $T_A$ started with the number of the most recently committed transaction $T_0$, $T_A$ will only find a version that exactly matches its transaction number if $T_0$ committed to that box. When that is not the case, $T_A$ uses the closest older body available in that box, which in the “worst case” will be the first version of the box.

Figure 4.1 shows a `VBox` holding an integer that was created by transaction 5 with the value 0 and then incremented by transactions 23 and 87. When reading from this box, a transaction running with the number 50 would skip the first body when traversing the list because it is...
tagged with a higher version number. In the absence of a value tagged exactly with the number 50, this transaction would then use the value 1, corresponding to version 23, because it is the highest version available with a number lower than the transaction’s version.

4.3.2 Read Set

Read sets are stored as a list of VBox arrays, implemented using cons pairs. By default, each array has a capacity of 1,000 boxes.

Whenever a transaction reads a value from a box, JVSTM places a reference to that box in the first array (the “active” array) of the list. An integer field (next) keeps track of the next position in the active array to be used. When this value is below zero, JVSTM pushes a new, clean array into the top of the list and resets the next field so that further reads are stored in the new array.

Additionally, JVSTM keeps a pool of arrays that is local to each running thread. When a
4.3. IMPORTANT DATA STRUCTURES

A transaction needs a clean array for its read set, it looks for one in the pool first and only creates a new one if the pool is empty. When the transaction finishes, it returns all the arrays it used to the pool, so that the next transaction that runs in the same thread can reuse them. To further save time, transactions do not need to clear their arrays before returning them to the pool, as future transactions can simply overwrite their contents.

The example in Figure 4.2 shows the read set of a transaction that performed 2,500 reads — earlier reads are stored towards the end of the list and the first array, containing the boxes read most recently, still has free space.

Because the active array is always the first one of the list and the next field provides direct access within said array, adding a box to the read set is very fast, in accordance with one of JVSTM’s most important requirements, namely that “accessing an object for reading should have a low overhead” (Cachopo 2007).

The simplicity of the insertion algorithm has the disadvantage of allowing duplicates — if a transaction reads from the same box repeatedly (inside a loop for example), it will also be repeatedly added to the read set. While not being very typical, such a scenario would lead to wasted space (due to the duplicate entries) and wasted time in the validation phase (where each entry of the read set is checked). Also, the simplicity of this structure also does not allow for an efficient lookup algorithm — because the contents are completely unsorted, the only way of looking up a box in the read set is by performing a sequential search, which is indeed what is used in JVSTM’s incremental validation algorithm.

4.3.3 Write Set

In JVSTM, a transaction’s write set is backed up by one structure during execution and is converted into a different structure when it reaches the validation phase.

4.3.3.1 During Execution

In a write transaction, the setBoxValue method, responsible for tentatively writing a value into a box, may opt to record write operations in one of two fields: boxesWritten or boxesWrittenInPlace. The former is a Java HashMap that maps a box to the tentative value written by the transaction and the latter is a list of box references implemented using cons pairs.

To decide where to put a reference to the box being handled, the setBoxValue method uses the currentOwner and orec fields (from the box and the transaction, respectively), which are ownership records. The OwnershipRecord class essentially has one integer field, called version, that indicates whether the transaction associated to that ownership record is running (version = 0), aborted (version = −1) or committed (version > 0).

When a transaction is created, its orec field is initialized with a unique ownership record. When boxes are created, their ownership field is initialized with the default value of 1. When a running transaction T attempts to write into a box, one of the following scenarios can happen:

- If the box’s ownership record is the one that belongs to T, T is guaranteed to be the box’s owner, meaning it has previously written into the box’s tempValue field. In this case,
JVSTM merely needs to overwrite that slot with the new value, as the box has already been added to T’s write set when it wrote there previously.

- If the box’s current owner is another transaction that is either still running or has already committed with a version number more recent than T’s, T cannot gain ownership of this box (and cannot use its temporary slot to store the new value) and thus JVSTM uses the boxesWritten map to record the write operation.

- Otherwise, if the box’s owning transaction is either aborted, committed with a version at most equal to T’s or if the box was never written by any transaction, T attempts to gain ownership of the box by performing a CAS to place its ownership record in the box’s currentOwner field. If the CAS fails, another transaction gained ownership of this box and thus the new scenario is reevaluated using the same rules. Otherwise, T writes the tentative value into the box’s tempValue field and stores a reference to the box in the boxesWrittenInPlace list.

In summary, a transaction’s write set is split into two parts: the boxes where it was able to write directly into the tempValue field (boxesWrittenInPlace) and the boxes where it had to record the tentative value locally (boxesWritten). This division has the goal of speeding up scenarios where the same transaction writes into the same box more than once and where it reads from a box after having written into it.

Using the aforementioned set of rules, it is possible that when a transaction writes into a box for the first time, the value is recorded in boxesWritten because it failed to gain ownership of the box, but in a second write operation on the same box, the same transaction may succeed to gain ownership and thus use the boxesWrittenInPlace list to record the second write. However, the transformation of the write set into a new structure described in the next section makes sure that only the correct value is written back during commit.

The use of a HashMap for part of the write set eliminates potential duplicates, as it allows to quickly find and overwrite a tentative value previously written by the same transaction to a given box, which is important because only the most recent value written by a transaction should be made permanent whenever it commits. Such consideration is unnecessary for the boxesWrittenInPlace because the tentative values of those boxes are not stored within this structure, but in the boxes themselves. Additionally, duplicates are avoided by only adding a box to this list when ownership is obtained and not in consecutive writes.

### 4.3.3.2 After Execution

In the lock-free implementation of JVSTM, all transactions that have finished running but are still waiting their turn in the commit queue take part in the process of writing back other transactions’ write sets. Because several transactions will be writing back the same write set, it needs to be converted to a structure that combines both previous structures used during execution (the boxesWritten map and the boxesWrittenInPlace list depicted in Figure 4.3) and allows the write-back work to be divided between transactions while minimizing duplicate work.

This conversion happens right after the transaction finishes running and just before it tries to enqueue itself in the Active Transactions Record. The new write set (Figure 4.4) is stored in
4.3. IMPORTANT DATA STRUCTURES

Figure 4.3: Example of a write set during execution

the respective transaction’s record and may be accessed by other transactions for write-back and validation purposes.

Figure 4.4: Example of a write set after execution

To combine both previous structures, the WriteSet class contains two arrays — one with references to all boxes where the transaction wrote and another with the values to be written in those boxes. In case the same box is present in both previous structures, its box/value pair from the boxesWrittenInPlace list takes precedence over the one from the map, as it is guaranteed to contain the most recent value of the two — once a transaction gains ownership of a box, it cannot lose it until it commits or aborts, so in case of conflict we know that the value written under ownership is the most recent.

These arrays allow direct access when writing back the write set, and thus each helping transaction can be assigned a block of boxes to write-back (the default block size is 10 boxes). Additionally, an array of atomically updated booleans keeps track of which blocks have been written-back.

As previously mentioned, this write set can be used for two purposes — validation and write-back. For the former, transactions simply need to go through the box array sequentially to perform incremental validation.

To take part in the write-back phase, a helping transaction chooses a starting block at random so that not all transactions start at the same block. Then it processes all blocks from that point on sequentially until it arrives back at the starting block, following the same protocol for every block:

- If the block is already marked as done by some other transaction, skip to the next one.
Otherwise, write-back all boxes of that block and mark it as done.

Because the atomic boolean variable that stores a block’s status is only changed after all boxes of that block have effectively been written-back, it is possible that a second transaction visits a block while another one is already writing back its values, thus resulting in duplicate work. The most obvious way of avoiding this would be for the first visitor to change that block’s status immediately before writing it back, but this would not be safe — if the thread of the helping transaction were to stop running during write-back, other transactions would wrongly see the block as fully written-back.

Figure 4.5 shows a WriteSet consisting in ten blocks and two helping transactions ($T_A$ and $T_B$). $T_A$ starts committing the boxes in block 2 and $T_B$ those in block 4. When $T_A$ eventually reaches block 4, it will most likely observe that it is already committed and will start skipping all blocks committed by $T_B$ until it reaches an uncommitted block. After processing the last block, number 10, both transactions go to block 1 and continue from there until each of them reaches its starting block again (2 and 4, respectively).

Despite not fully avoiding it under certain circumstances, the block status array still helps reduce duplicate work by letting helping transactions skip over some blocks safely. After visiting all blocks and reaching the starting point again, an helping transaction knows that all blocks have been successfully written-back (either by itself or others) and thus it can move on to helping the next transaction in the commit queue.

### 4.3.4 ActiveTransactionsRecord

In JVSTM’s original lock-based version, the Active Transactions Record (ATR) was essentially a list of active transactions kept for garbage collection purposes. It is implemented as simple linked list and each record therein contains information about a previously committed write transaction to help determine when the values it wrote can be safely garbage collected. Each record has 4 fields: the number of the transaction it refers to (transactionNumber), a list of the bodies written by that transaction (bodiesToGC), the number of transactions that are still running under the same version number (running) and a reference to the next record (next).

The lock-free version of JVSTM extends the ATR’s functionality — transactions enter the ATR as soon as they are successfully validated and records now also hold their respective trans-
4.4 Performance Evaluation

4.4.1 Statistical Data Collection

Before attempting to optimize JVSTM’s scalability, it is important to understand the impact of each phase of a transaction’s lifecycle in the current implementation. To that end, I extended JVSTM’s existing CommitStats class to collect data not only for the commit phase, but for all others as well, namely:

1. Transaction type (Read-only, read/write, aborted, etc.)
2. Begin phase
   - Duration of the begin phase (initialization of structures, etc.)
3. Execution phase
   - Time spent reading from boxes
   - Time spent writing into boxes
4. Commit phase
   - Time spent helping other transactions commit
• Time spent in snapshot validation
• Time spent in incremental validation
• Number of incremental validations performed
• Time spent writing back values
• Time spent cleaning up
• Duration of the abort process (for aborted transactions)

After collecting this data, TxStats is able to display it either individually per transaction or globally as average values per transaction type.

It is also important to note that TxStats collects data by introducing additional instructions into JVSTM, similarly to what profilers do, and thus it has a negative impact on performance. The size of the impact depends on different parameters such as workload and number of concurrent transactions, which makes it impossible to define a specific measure of the impact of data collection.

4.4.2 Test Environment

4.4.2.1 Hardware

All test results described in this and the following chapters of this work were made using a server with 4 AMD Opteron 6168 1.9 GHz processors, each with 12 cores, totaling 48 hardware threads. The total RAM of the server is 128 GB, with the JVM (version 1.6) being assigned a maximum of 16 GB.

4.4.2.2 Benchmark

To evaluate JVSTM’s performance in its original state and with the improvements proposed in the next chapter, I used a simple micro-benchmark created by JVSTM’s developers. This benchmark consists in an array of versioned boxes that is shared among transactions. Each box encapsulates an Integer that initially has the value 0. Each transaction performs several read and/or write operations on boxes — for each operation a random number is chosen to decide on which box of the shared array the operation will be performed. Write operations consist in writing the same value as the box’s position in the array and read operations copy the box’s value to a local variable.

To allow the creation of different test scenarios, the benchmark allows the customization of the following (and other) parameters:

• Size of the shared data array (number of versioned boxes).
• Number of simultaneous threads (each thread executes a transaction).
• Number of transactions to run.
• Percentage of write-only transactions.
• Percentage of read/write transactions.
4.4. PERFORMANCE EVALUATION

- Number of write operations in a write-only transaction.
- Number of read operations in a read-only transaction.
- Number of read operations in a read/write transaction.
- Number of write operations in a read/write transaction.

Because read and write operations are performed in a random order and in random boxes, there is no guaranteed relation between the boxes read and written. This lead me to change read/write transactions to only write into boxes that they previously read, as in my experience this is a better representation of a typical object-oriented application's behavior.

4.4.3 Early Conclusions

To evaluate JVSTM's current performance and set a baseline for the optimizations proposed in the next chapter, I used the test environment described above with different parameters to collect data under different scenarios and understand which phases are more relevant in each of them. The following parameters were used:

- Size of the shared data array: 1,000,000 boxes.
- Number of simultaneous threads: 1, 2, 4, 8, 16, 32, 48, 64, 96.
- Number of transactions to run: 250,000.
- Percentage of write-only transactions: 0%.
- Percentage of read/write transactions: 0%, 100%.
- Number of read operations in a read-only transaction: 1,000.
- Number of read operations in a read/write transaction: 500, 900, 990.
- Number of write operations in a read/write transaction: 10, 100, 500.

The combination of these parameters results in a multitude of scenarios that allows us to observe different strengths and weaknesses of the JVSTM.

The analysis of how JVSTM works, in addition to the statistical data collected in different test scenarios, allowed to identify the following areas where JVSTM can potentially be improved by changing or tweaking structures or algorithms:

- Due to its structure, the read set can only be searched sequentially. An improved structure could speed up the incremental validation process.
- The two-phase validation process is always the same regardless of how many transactions are pending in the commit queue. A dynamic solution that applies different strategies defined by the current scenario could perform better.
- Validation could be performed collaboratively, similarly to what happens in the write-back phase.
• Even though the write-back help algorithm attempts to reduce duplicate work, additional limiting measures can be attempted.

• When a conflict is found, no attempts are made to “save” the transaction and the work it has already performed.

In some cases, JVSTM’s functionality was extended to also write back transactions’ values to disk, which generates different challenges and opens up different optimization possibilities.
In this chapter I will describe the several ideas I implemented to attempt to improve JVSTM’s scalability.

For each idea I describe in what it consists, how it changes the JVSTM, what benefits are expected a priori and, whenever possible, any foreseeable drawbacks. For all implementations I will provide two basic metrics, total running time and number of aborted transactions, to evaluate how they differ from the original version of the JVSTM. Whenever those two parameters do not represent the expected behaviors, I will also present more detailed parameters obtained using the statistics class described in the previous chapter in order to be able to explain the unexpected behavior.

In some cases, the interpretation of those parameters will be followed by a variation of the idea that attempts to solve any flaws found in the original rationale that were discovered by the results. For those variations I will also provide a description, expected benefits and a comparison of its results and those of the original idea or the original implementation of the JVSTM.

To allow for the ideas’ results to be clear and easily comparable with each other, all implementations presented here will be tested using the same benchmark described in the previous chapter. Due to the overhead introduced by the TxStats statistics collection class described previously, when comparing the performance of different implementations in this chapter I will only present the values for running time and number of aborted transactions collected with statistics turned off, to guarantee that the results are not tainted by factors that do not concern the optimizations themselves. However, when the performance results of an implementation require a more in-depth analysis, I will present some parameters collected with statistics turned on — naturally, those values will come from a different test run and thus some discrepancies to the performance results can occur.
CHAPTER 5. PROPOSED OPTIMIZATIONS

5.1 Implement the Read Set Using an Identity Hash Set

5.1.1 Description

To keep track of each transaction’s read set, JVSTM uses a list of arrays to store references to each box that the transaction reads. The simplicity of this structure makes it very fast to insert a new box in the read set, in observance of JVSTM’s requirement of low overhead read accesses.

Lookup operations are very simple too, but also expensive because they require a linear search, with a worst-case cost of $O(n)$ (where $n$ is the size of the read set). This cost is of great significance in the incremental validation process, which must guarantee there is no intersection between the validating transaction’s read set and another transaction’s write set. To do this, each box in the given write set must be looked for in the other transaction’s read set — in the absence of a conflict between both transactions, which is the most desired scenario in an STM, all these lookups will represent the worst-case scenario in terms of cost. This means that the worst-case cost for a single run of the incremental validation algorithm is $O(m \times n)$, with $m$ being the size of the write set, but a single transaction may make several runs ($k$) of this validation process. Thus, the total complexity of incremental validation for one transaction is $O(k \times m \times n)$, where $m$ is the average size of the $k$ write sets. For a validating transaction $T$, $k$ is the number of transactions that were validated and entered the commit queue between $T$ having started snapshot validation and successfully entering the queue itself — from this follows that $k$ grows as the number of concurrent transactions rises, because increased concurrency will lead to more transactions entering the queue in a given timeframe.

To speed up the lookup process, I propose to use an Identity Hash Set (IHS) as a substitute for the list of VBox arrays currently in use to store the transaction’s read set. Essentially, the IdentityHashSet class has the same properties as a traditional Java HashSet, except all key comparison operations are implemented using reference-equality instead of object-equality, in an effort to make them as fast as possible and reduce overhead. The use of reference-equality is possible in JVSTM’s context because we know that the only way for two VBox objects to be equal is if they are one and the same. In an effort to be even more efficient, the hash function does not rely on remainder operations as Hashtable does. Instead, the IdentityHashSet’s size is always set to a power of 2, so that the hash function can be implemented using a bitwise AND operation. The code for the IdentityHashSet class, an adaptation from Java’s Hashtable source code, can be found in Appendix A.

This new structure allows for much faster lookups because the read set is split in several buckets and each lookup will only be done within the bucket determined by the hash function. Moreover, its different insertion process also avoids duplicate entries, which would be too costly to guarantee using the current array implementation. This will yield a time and space advantage in scenarios where one box is read $n$ times by the same transaction — with the IdentityHashSet the VBox reference will only be stored once, whereas in the current implementation it would be stored $n$ times.

However, it is also in the insertion algorithm that the IdentityHashSet’s biggest disadvantage lies. With JVSTM’s current implementation, inserting a box in the read set consists only in putting its reference at the end of the current array, and eventually creating a new array if the current one is full. Inserting a new item in an IdentityHashSet, though, requires several more steps:
1. Determine the bucket by calculating the item’s hash value;
2. Check if the item is already in the bucket (if so, stop, otherwise proceed);
3. If the set is growing too big, resize it by rehashing and reinserting all elements;
4. Insert the new item.

Step 2 could be made redundant by the choice to allow duplicate entries, but doing it would make the set grow more, thus leading to more resize operations. Also, the increased number of elements inside the same bucket caused by these duplicate entries could lead to slower lookups done on the bucket. The resize operation described in Step 3 behaves like in any traditional hash table: when a given threshold is reached, a new bucket list must be created and all elements must be rehashed and reinserted all over again, making this a very costly operation, hence why duplicate entries should not be allowed.

5.1.2 Initial Results

To test this idea, I used a suite of 9 tests, ranging from 1 to 96 simultaneous threads, where each thread executes one transaction at a time; each test was composed of 250,000 transactions that executed 1,000 operations each, with 10% being write operations, meaning each transaction read 900 boxes and wrote into 100, all randomly picked from a universe of 1,000,000 shared boxes.

<table>
<thead>
<tr>
<th>Threads</th>
<th>Original JVSTM</th>
<th>JVSTM with IdentityHashSet</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time</td>
<td>Aborts</td>
</tr>
<tr>
<td>1</td>
<td>95.125</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>47.247</td>
<td>9.605</td>
</tr>
<tr>
<td>4</td>
<td>25.848</td>
<td>27.807</td>
</tr>
<tr>
<td>8</td>
<td>15.326</td>
<td>64.839</td>
</tr>
<tr>
<td>16</td>
<td>14.118</td>
<td>128.357</td>
</tr>
<tr>
<td>32</td>
<td>13.569</td>
<td>239.842</td>
</tr>
<tr>
<td>48</td>
<td>13.142</td>
<td>340.846</td>
</tr>
<tr>
<td>64</td>
<td>13.097</td>
<td>348.738</td>
</tr>
<tr>
<td>96</td>
<td>13.373</td>
<td>363.833</td>
</tr>
</tbody>
</table>

Table 5.1: Total running time (in milliseconds), number of aborted transactions and number of transactions committed/aborted per millisecond in the original version of the JVSTM and in the version with IdentityHashSet (250,000 txs, 1,000 ops/tx, 10% write ratio)
Figure 5.1: Running time

Figure 5.2: Number of aborted transactions
most of the times there is just a single transaction trying to commit. The IdentityHashSet improves this stage but it also incurs in an extra cost during the execution of a transaction since the cost of inserting a box in the read set is higher for an hash set than for a list of arrays. This explains why the IdentityHashSet is slower for low concurrency tests. When the concurrency level is higher than 8, then the probability of having several transactions trying to commit concurrently is not negligible and the original version will spend a long time doing incremental validation. In these scenarios, the IdentityHashSet performs incremental validation faster, and that translates into an improvement between 27% and 42%.

The number of aborted transactions is more intriguing because it remained almost the same up to 8 threads in both versions and then diverged up to 29% in the following tests. From this follows that in the high-concurrency tests (16 threads and more) JVSTM commits more transactions per second when using the IdentityHashSet. If we also factor in aborted transactions, the difference in number of transactions processed per second is even higher. However, these are unexpected results that need further data to be explained — intuitively, one would expect that the IdentityHashSet would speed up incremental validation without impacting the number of aborted transactions (at least as significantly as the results show).

To understand this behavior I analyzed how aborts are distributed by each stage (remember that in JVSTM transactions can abort either while reading a value during execution or by failing validation, either snapshot or incremental). These results can be seen in Table 5.2. First, note that the results in this table are from a different test run (with collection of detailed statistics) than those in Table 5.1 (without statistics) and as such the number of aborts is different in both tables despite the tests having the same parameters. Furthermore, the sum of all three percentages of a particular test is not always exactly 100% because they are rounded to the nearest integer. However, neither of these factors prevents us from observing how aborts are distributed.

<table>
<thead>
<tr>
<th>Threads</th>
<th>Original JVSTM</th>
<th>JVSTM with IdentityHashSet</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Read</td>
<td>Snapshot</td>
</tr>
<tr>
<td>1</td>
<td>0 (−)</td>
<td>0 (−)</td>
</tr>
<tr>
<td>2</td>
<td>7.632 (80%)</td>
<td>1.728 (18%)</td>
</tr>
<tr>
<td>4</td>
<td>22.383 (80%)</td>
<td>4.862 (17%)</td>
</tr>
<tr>
<td>8</td>
<td>45.261 (69%)</td>
<td>11.272 (17%)</td>
</tr>
<tr>
<td>16</td>
<td>60.515 (43%)</td>
<td>15.686 (12%)</td>
</tr>
<tr>
<td>32</td>
<td>83.648 (33%)</td>
<td>23.952 (9%)</td>
</tr>
<tr>
<td>48</td>
<td>97.455 (26%)</td>
<td>30.099 (8%)</td>
</tr>
<tr>
<td>64</td>
<td>104.894 (28%)</td>
<td>30.606 (8%)</td>
</tr>
<tr>
<td>96</td>
<td>117.854 (31%)</td>
<td>31.181 (8%)</td>
</tr>
</tbody>
</table>

Table 5.2: Number of aborted transactions in each stage in the original version of the JVSTM and in the version with IdentityHashSet (250,000 transactions, 1,000 operations/transaction, 10% write ratio)

In this table we can see that, starting at 16 threads, there is a clear shift of aborted transactions from incremental validation to the earlier stages, especially to read operations — for example, in the 96 threads test, aborts during read operations and snapshot validation in JVSTM with IHS more than doubled, while aborts during incremental validation are down to less than a third (from 231.190 in the original version to 74.874 with IHS. This considerable shift in distribution explains why the IdentityHashSet allows JVSTM to perform faster in these tests.
despite the number of aborted transactions being much higher, as observed previously in Figure 5.2.
In the original version, most (up to $\frac{2}{3}$) transactions were aborting in the latest stage possible — incremental validation —, which meant they had already wasted time executing and performing snapshot validation before eventually failing at incremental validation. This happened because in this implementation incremental validation was considerably slower, causing more pending transactions to accumulate in the commit queue, which in turn leads to an increased probability of transactions aborting in the incremental validation phase.

On the contrary, with the IdentityHashSet most transactions (at least $\frac{2}{3}$) are aborting at the earliest stage possible — a read operation during execution —, which means they do not waste time finishing execution and performing one or both validation steps. This time both compensates for the increased number of aborts of this solution and simultaneously allows these tests to be faster.

Regarding the lower-concurrency tests (up to 8 threads), their abort distribution was already balanced heavily towards the earliest stage in the original version of the JVSTM, and that did not change with IdentityHashSet. This is coherent with the fact that the number of aborts in these tests is very similar in both versions.

Now that we have established why high-concurrency tests are faster despite the increased number of aborts, we need to understand the reasons behind the different abort distribution and why the overall number of aborts increased. As we have seen in Table 5.1, more transactions are committed per second with the IdentityHashSet than without. A direct implication of this is that more boxes are updated in the same time period, which in turn increases the probability of any transaction aborting during a read operation or snapshot validation — remember that transactions abort in one of these two stages if a box is updated with a version newer than the transaction’s number.

![Figure 5.3: Average execution time of a committed transaction (in microseconds; 96 threads, 250,000 transactions, 1,000 operations/transaction, 10% write ratio)](image)

However, there is another factor contributing to the increase of aborted transactions. Figure 5.3 shows that the average time a successful (committed) transaction spent in the execution phase in the 96 threads test is greater in the version with the IdentityHashSet. The increased time in this phase is due to the IdentityHashSet’s disadvantage when inserting items, as
explained in Section 5.1.1.

Finally, in Table 5.3 we can see that, while in the IdentityHashSet implementation the number of transactions (both successful and aborted) that perform incremental validation is higher than before, it is also true that, on average, each transaction performs less incremental validation steps. Also, as expected, the use of the IdentityHashSet structure leads to a considerable reduction of the time that each incremental validation step requires. Ultimately, this leads to faster validation and less “congestion” in the commit queue, which explains why fewer transactions abort during incremental validation. In turn, the faster validation process means that the values of successful transactions are also written back faster, which, as we have seen previously, is the reason behind the increase in transactions that abort during read operations.

<table>
<thead>
<tr>
<th>Threads</th>
<th>Original JVSTM</th>
<th>JVSTM with IdentityHashSet</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0.0</td>
</tr>
<tr>
<td>2</td>
<td>7.621</td>
<td>1.0</td>
</tr>
<tr>
<td>4</td>
<td>25.000</td>
<td>1.2</td>
</tr>
<tr>
<td>8</td>
<td>70.707</td>
<td>3.6</td>
</tr>
<tr>
<td>16</td>
<td>104.853</td>
<td>14.7</td>
</tr>
<tr>
<td>32</td>
<td>177.451</td>
<td>21.9</td>
</tr>
<tr>
<td>48</td>
<td>277.232</td>
<td>23.8</td>
</tr>
<tr>
<td>64</td>
<td>267.987</td>
<td>24.0</td>
</tr>
<tr>
<td>96</td>
<td>258.392</td>
<td>24.0</td>
</tr>
</tbody>
</table>

Table 5.3: Number of transactions that perform incremental validation, average number of incremental validation steps performed by each transaction and average time per single validation (250.000 transactions, 1.000 operations/transaction, 10% write ratio)

In summary, we have concluded the following about JVSTM’s behavior with IdentityHashSet in high-concurrency scenarios:

- Performance is better (the same number of transactions is committed in less time).
- The number of aborted transactions is higher, but because transactions abort much earlier, less resources are wasted overall.
- Because aborts are faster, threads are freed earlier to run other transactions. This results in more transactions overlapping in time, increasing the probability of aborts.
- The execution time of transactions is longer, which increases overlap (and thus abort probability) further.
- Despite the number of transactions that perform incremental validations having increased, each one performs less incremental validation steps and each step is also faster.

In low-concurrency scenarios, transactions already aborted in earlier stages, and thus the same gains are not observed. However, in these tests transactions still suffer from IdentityHashSet’s disadvantage of slower insertion times, which explains why JVSTM with IdentityHashSet is slower in these scenarios.
5.1.3 Variation with a Hybrid Structure for the Read Set

As seen in the previous experiment that always uses an IdentityHashSet as a structure for transactions’ read sets, the balance between cost and benefit of the IdentityHashSet structure determines a positive or negative outcome depending on the circumstances. In this section I present the results of a variation of the JVSTM that attempts to combine the benefits of both the original array solution and the IdentityHashSet.

As we have seen previously, IdentityHashSet introduces a higher cost in the insertion operation than the simple array structure used before, but the benefits are only reaped if the transaction performs incremental validations against other transactions, which does not always happen. To try to avoid those costs in some situations, this adaptive solution uses the original array structure as the default for the read set during transaction execution, and postpones the creation of the IdentityHashSet until after snapshot validation. Thus, transactions that successfully enter the commit queue immediately after snapshot validation will not incur the extra costs of the IdentityHashSet. In turn, transactions that do end up creating and populating an IdentityHashSet will need more time and space than in the non-adaptive version because they will have populated the traditional read set before converting it into the new structure, but hopefully this extra cost will be offset by the benefits of using IdentityHashSet.

![Figure 5.4: Running time in the original version, IdentityHashSet and hybrid read set (in seconds; 250,000 transactions, 1,000 operations/transaction, 10% write ratio)](image)

Figure 5.4: Running time in the original version, IdentityHashSet and hybrid read set (in seconds; 250,000 transactions, 1,000 operations/transaction, 10% write ratio)

Table 5.4 shows the running time and number of aborted transactions for the original version of the JVSTM, the one that uses IdentityHashSet exclusively and the one with a hybrid read set for the tests with 250,000 transactions, 1,000 operations per transaction and a 10% write ratio. Figures 5.4 and 5.5 offer a graphical representation of these same values. Comparing the performance of the IHS version with the hybrid read set, we can see that tests up to 8 simultaneous threads are significantly faster (11 – 19%), since in this situation most of the times it is not useful to use an IdentityHashSet for the read set (as the probability of doing incremen-
5.1. IMPLEMENT THE READ SET USING AN IDENTITY HASH SET

Figure 5.5: Number of aborted transactions in the original version, IdentityHashSet and hybrid read set (250,000 transactions, 1,000 operations/transaction, 10% write ratio)

<table>
<thead>
<tr>
<th>Threads</th>
<th>Running time (ms)</th>
<th>Aborted transactions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Original</td>
<td>IHS</td>
</tr>
<tr>
<td>1</td>
<td>91.358</td>
<td>99.949</td>
</tr>
<tr>
<td>2</td>
<td>47.247</td>
<td>55.079</td>
</tr>
<tr>
<td>8</td>
<td>15.326</td>
<td>17.020</td>
</tr>
<tr>
<td>16</td>
<td>14.118</td>
<td>10.320</td>
</tr>
<tr>
<td>32</td>
<td>13.569</td>
<td>7.917</td>
</tr>
<tr>
<td>64</td>
<td>13.097</td>
<td>8.339</td>
</tr>
<tr>
<td>96</td>
<td>13.373</td>
<td>8.652</td>
</tr>
</tbody>
</table>

Table 5.4: Running time and number of aborted transactions in the original version, IdentityHashSet and hybrid read set
tal validation is low). Higher concurrency tests only show marginal variations (±5%) because in this case there is incremental validation most of the times. The number of aborted transactions shows a similar behavior to the time results — low-concurrency tests perform similarly to how they did in the original version, whereas high-concurrency tests have values similar to the IdentityHashSet version. Comparing the hybrid read set with the original implementation, it matches JVSTM’s performance under low concurrency (up to 8 threads) with slight performance gains, and performs clearly better than the original implementation with 16 or more threads.

The conclusion is that low-concurrency tests perform as well as in the original version, while high-concurrency tests perform similarly to the version with the IdentityHashSet. This means that, in effect, the hybrid read set succeeded in combining the strengths of both versions, as was expected.

5.1.4 Results in other Scenarios

Having drawn some conclusions on IdentityHashSet’s impact in a scenario where each transaction had a 10% write ratio, and having perfected those results with the hybrid read set solution, let us now look at how the system performs with different ratios.

5.1.4.1 1% Write Ratio

![Figure 5.6: Performance of the original version, IdentityHashSet and hybrid read set (250,000 transactions, 1,000 operations/transaction, 1% write ratio)]

Figure 5.6 shows that, with a 1% write ratio, the IdentityHashSet alone performed noticeably worse than the original version (up to 33% slower). The hybrid read set performed better by matching the original version’s results up to 16 threads, but was still up to 15% slower in tests with more concurrency. In either version, the number of aborted transactions remained unchanged with variations in the ±3% range.

To understand why this happens, let us analyze how aborts are distributed by the transaction’s lifecycle in this scenario for all three versions. Remember that in Section 5.1.2 we observed that in the original version of the JVSTM, aborts were distributed more heavily towards the end of the transaction’s lifecycle (incremental validation) and that using the
IdentityHashSet changed the bias towards an earlier point (read operations), thus reducing the total amount of work wasted in aborted transactions, which translated into better overall performance.

![Graph showing abort distribution between original version, IdentityHashSet and hybrid read set](image)

**Figure 5.7:** Abort distribution between original version, IdentityHashSet and hybrid read set (250.000 transactions, 1.000 operations/transaction, 1% write ratio, 96 threads)

However, in this scenario we have smaller write sets (10 instead of 100), which means that the ratio between the time required to validate/commit a transaction and the time the transaction spent running is smaller (Figure 5.8 versus Figure 5.3). This translates into a higher probability of aborting during execution rather than validation, even in the original implementation, as can be seen in Figure 5.7. In fact, the IdentityHashSet implementation barely changed the already favorable distribution, and the hybrid read set proposal actually made it worse, which is why these implementations have worse performance than the original one with...
a 1% write ratio. But, if the IdentityHashSet’s abort distribution is similar to the original version’s and the hybrid read set’s distribution is worse, why did the latter have better results than the IdentityHashSet implementation? Notice the difference between both implementations’ average transaction times in Figure 5.8 — as we have concluded previously, using the IdentityHashSet introduces extra costs during the transactions execution. Due to its nature, the hybrid read set implementation only creates an IdentityHashSet under certain conditions, and because of that it is not as exposed to its disadvantages. Regardless, neither implementation’s benefits are enough to offset the extra costs in this scenario, which leads us to conclude that they are not suitable for workloads where transactions have low write ratios.

5.1.4.2 50% Write Ratio

![Graphs](image)

(a) Running time (in seconds)  (b) Number of aborted transactions

Figure 5.9: Performance comparison between original version, IdentityHashSet and hybrid read set (250,000 transactions, 1,000 operations/transaction, 50% write ratio)

With a 50% write ratio (Figure 5.9), IdentityHashSet performed worse than the original version under low concurrency (up to 10% slower), was 9 – 11% faster in the middle tests (16 and 32 threads) and had similar performance under high concurrency (3% slower in tests with 48 threads or more). The number of aborted transactions had the same behavior that we already observed in the 10% scenario — it was similar to the original version under low concurrency and then increased up to 35% in other tests.

The hybrid read set performed better — under low concurrency it was as fast (±3%) as the original version, and in tests with 8 threads or more it was between 7% and 15% faster. The number of aborted transactions increased up to 16% — while still significant, it is a less dramatic difference than in the IdentityHashSet version.

To better understand these results, let us again look at how aborts are distributed throughout the transaction’s lifecycle in the three implementations. The distribution shown in Figure 5.10 refers to the 96 threads test, where the IdentityHashSet version was 3% slower and the hybrid read set implementation 10% faster than the original.

The first thing to note is that, unlike the 1% scenario, the majority of aborts in the original version happen in the incremental validation phase, which means that there is room for improvement, as with the 10% ratio. In fact, the abort distribution of both alternative implementations are clearly better than the original’s, but it is also worth noting that the improvements
5.1. IMPLEMENT THE READ SET USING AN IDENTITY HASH SET

![Bar chart showing abort distribution between original version, IdentityHashSet, and hybrid read set](image)

Figure 5.10: Abort distribution between original version, IdentityHashSet and hybrid read set (250,000 transactions, 1,000 operations/transaction, 50% write ratio, 96 threads)

are not as impressive as in the 10% tests — as we have seen in the same test with the 10% write ratio in Table 5.2, IdentityHashSet changed a 31%/8%/61% distribution to 67%/16%/17%, whereas here we still have more than 60% of transactions aborting during both validation steps. Secondly, it is also worth noting that in this scenario the hybrid read set did not increase the total number of aborts as much as the IdentityHashSet, whereas with the 10% ratio we have observed similar abort numbers in both implementations.

In the case of the IdentityHashSet, the less impressive improvement in abort distribution (compared to the 10% scenario) is only enough to offset the increased number of aborted transactions associated with this structure, which is why this solution has roughly the same performance as the original. Having a smaller number of aborted transactions, the hybrid read set is successful in offsetting that cost with more aborts happening earlier.

5.1.4.3 100% Write Ratio

Finally, in the scenario with a 100% write ratio, the time results of all three versions are identical and the number of aborted transactions is also always unchanged at 0. Because having a 100% write ratio means having no read operations, the transactions in these tests never add anything to their read sets, and validation is always successful because empty read sets cannot generate conflicts. Thus, any impacts that alternative read set structures have on validation are not reflected in this scenario.
5.2 Adaptive Validation

5.2.1 Motivation

When a transaction $T$ completes the execution phase, it must be validated before entering the Active Transactions Record (ATR). As explained in Section 4.2.3, the validation process has two phases: snapshot and incremental. The first one is mandatory and consists in checking every box in $T$’s read set for updates. If this succeeds, JVSTM attempts to put $T$ in the ATR, which may trigger incremental validations against other transactions $U_{1..N}$. Validating $T$ incrementally against a transaction $U_x$ consists in looking for intersections between $U_x$’s write set and $T$’s read set. This allows the validation process to be lock-free and still guarantee a valid commit order for concurrent transactions. The rationale behind this two-phase process is that performing incremental validation should be cheaper than repeating snapshot validation.

Let us consider the scenario where a transaction $X$ tries to enter the ATR only 4 slots after it started (when a transaction starts, its activeTxRecord field is set to the record of the most recently committed transaction, which will define the context under which the new transaction will run). Let us also assume all transactions in this scenario read 900 boxes and attempt to write into 100. As shown in Figure 5.12, when $X$ tries to enter the ATR, there will be 3 records between its activeTxRecord and the ATR’s end. In this scenario, JVSTM would perform a snapshot validation, which would guarantee $X$ is valid up to transaction 6. Then, it would attempt to put $X$ in the ATR, eventually triggering incremental validations. Under these cir-
5.2. ADAPTIVE VALIDATION

cumstances, snapshot validating $X$ would require validating up to 900 boxes (its read set). However, we know upfront that $X$ is valid at least up to and including transaction 3, since this is $X$’s `activeTxRecord`. This means that $X$’s snapshot validation could be substituted with 3 incremental validations against transactions 4, 5 and 6, which equates to checking only 300 boxes (remember each transaction wrote into 100 boxes).

With this in mind, I propose that JVSTM adopts an adaptive strategy to choose whether to use the current two-phase validation scheme (with both snapshot and incremental validation) or only incremental validation. I implemented this strategy by calculating the difference between the transaction numbers of records `lastSeenCommitted` and `activeTxRecord` (both relative to transaction $X$), as shown in Listing 5.1. If this difference, represented by the `difference` variable, is below a certain threshold, only incremental validation is done starting at `activeTxRecord`, otherwise the two-phase process is performed.

```java
protected void validate() {
    ActiveTransactionsRecord lastSeenCommitted = helpCommitAll();
    int distance = lastSeenCommitted.transactionNumber - activeTxRecord.transactionNumber;
    if (distance < THRESHOLD) {
        // Incremental validations starting at activeTxRecord
    } else {
        // Snapshot validation
        // Incremental validations starting at lastSeenCommitted
    }
}
```

Code Listing 5.1: Code for adaptive validation

5.2.2 Initial Results

As for the previous ideas, let us start by analyzing the scenario with a 10\% write ratio.

Figures 5.13 and 5.14 show the results of this test suite measured, respectively, in running time and number of aborted transactions for both the original algorithm and the adaptive version as described in Section 5.2.1. The first test, where only one transaction is executed at a time and thus there is no concurrency, shows a marginal difference, whereas all tests with concurrency run between 22\% and 64\% slower, contrary to expectations. Aborts are also up by 10 – 28\% in the high-concurrency tests (with 16 threads or more), whereas the tests with 2 – 8 simultaneous threads have more modest increases of 1 – 2\%.

While the increased number of aborted transactions is the most obvious source of the extra execution time, further testing revealed another problem — the time saved in snapshot validations was less that the time added by the increased number of incremental validations.

Recalling how each type of validation works shows why the decrease from 900 to 300 boxes to validate given in the example in Section 5.2.1 did not translate into time savings. In the snapshot validation process, JVSTM would have to look into the 900 boxes in the read set and make a simple integer comparison to validate each one. By substituting this with 3 incremental validations as per the example, we now have to search for 300 boxes (the combined write sets of the 3 transactions) inside the read set of the transaction we are validating. Because JVSTM’s
CHAPTER 5. PROPOSED OPTIMIZATIONS

Figure 5.13: Running time

Figure 5.14: Number of aborted transactions
read sets are unsorted arrays, intersecting these sets will require up to 270,000 reference comparisons and thus takes longer than snapshot validation, contrary to what I initially thought. In the next section I propose a way of solving this problem.

Additionally, the value of the threshold that decides whether to use snapshot validation or not is key to the performance of adaptive validation. Intuitively, lower thresholds should yield better results — higher values will cause the substitution of one snapshot validation for several incremental validations, which, as least for the sizes of the read and write sets being considered in the micro-benchmark, is less likely to be an advantage. To test this theory, I experimented with the following values: 2, 3, 5 and 10. Notice that in Listing 5.1, the threshold is used as a non-inclusive upper bound, which means that the maximum distance between records in the commit queue for which snapshot validation will be skipped is 1, 2, 4 and 9, respectively. In other words, when using the lowest threshold, adaptive validation will only choose to skip the snapshot process when, just before beginning validation, a transaction does not observe any new records in the commit queue since it started. These tests confirmed that lower thresholds had the best overall results, and thus I will use the lowest value for all adaptive validation tests henceforth.

5.2.3 Variation with a Different Read Set Implementation

Having identified the structure of the read set as an impediment to the idea of adaptive validation, I decided to combine Adaptive Validation with the Adaptive IdentityHashSet implementation for the read set presented in Section 5.1.3. As described there, the hybrid IdentityHashSet structure provides faster lookup times, and thus is a prime candidate to solve this problem.

Continuing with the previous example, validating the same 300 boxes now means looking them up in the bucket corresponding to each box’s hash. Because each bucket will have only a small fraction of the read set, I expect the incremental validation will now require only a fraction of the previous 270,000 reference comparisons.

5.2.4 Results with Adaptive IdentityHashSet

Unlike what happened with Adaptive Validation by itself, this variant with the hybrid read set shows much improved results (Figure 5.15) in the 10% write ratio tests with 8 or more simultaneous threads: with 8 threads there is an 9% improvement in running time, whereas the tests between 16 and 96 threads are 28 – 38% faster. Unsurprisingly, these results display a behavior similar to that seen in Section 5.1.3 with the adaptive IdentityHashSet: low concurrency tests are slower and high concurrency tests show considerable gains, whereas the number of aborted transactions (Figure 5.16) also shows a comparable behavior. Thus, it is important to compare these results with those of the implementation with adaptive IHS and traditional validation, in order to assess the effectiveness of adaptive validation.

By observing Table 5.5, which shows the impact of adaptive validation in the total running time of the 10% write ratio scenario, it is clear that this technique has a positive impact when there is no concurrency and that it barely changes the results of tests with 4 simultaneous threads and above.
CHAPTER 5. PROPOSED OPTIMIZATIONS

Figure 5.15: Running time

Figure 5.16: Number of aborted transactions
5.2. ADAPTIVE VALIDATION

<table>
<thead>
<tr>
<th>Threads</th>
<th>Adaptive IHS</th>
<th>Adaptive IHS w/ Adapt. Valid.</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>102.207</td>
<td>87.132</td>
<td>15%</td>
</tr>
<tr>
<td>2</td>
<td>50.189</td>
<td>61.139</td>
<td>-22%</td>
</tr>
<tr>
<td>4</td>
<td>30.297</td>
<td>29.032</td>
<td>4%</td>
</tr>
<tr>
<td>8</td>
<td>17.051</td>
<td>16.919</td>
<td>1%</td>
</tr>
<tr>
<td>16</td>
<td>11.237</td>
<td>11.528</td>
<td>-3%</td>
</tr>
<tr>
<td>32</td>
<td>10.049</td>
<td>10.085</td>
<td>0%</td>
</tr>
<tr>
<td>48</td>
<td>10.129</td>
<td>10.449</td>
<td>-3%</td>
</tr>
<tr>
<td>64</td>
<td>10.279</td>
<td>10.201</td>
<td>1%</td>
</tr>
<tr>
<td>96</td>
<td>10.401</td>
<td>10.568</td>
<td>-2%</td>
</tr>
</tbody>
</table>

Table 5.5: Total running time of adaptive IHS and adaptive IHS with adaptive validation (in milliseconds, 10% write ratio)

This result can be explained by how the commit queue, the Active Transactions Record, works — as explained in Section 4.2.3, transactions enter the commit queue when they are guaranteed to be valid. Naturally, as the number of transactions running concurrently increases, the higher will be the number of records entering the commit queue between a given transaction starting and entering the queue. Thus, the distance between the activeTxRecord and lastSeenCommitted records is less likely to fall below the adaptive validation threshold as concurrency increases, while it will always fall below the threshold when there is no concurrency, which explains why the first test shows improvements and the high-concurrency ones barely show an impact.

However, this does not explain the results of the 2 thread test — intuitively, one would think that with 2 simultaneous threads there would be a 50% chance of a transaction falling below the threshold.

Finally, let us also compare this implementation with the original in the 1% and 50% scenarios. Figures 5.17 and 5.18 show results very similar to those of the original implementation, both in time and number of aborted transactions. The exceptions are the 1 and 2 thread tests, which are better and worse, respectively, as in the 10% scenario. Unsurprisingly, all other tests have results very similar to those observed previously without adaptive validation, as adaptive validation is very unlikely to change the behavior under high concurrency.

From these results we can conclude that adaptive validation can only be considered if the read set structure is altered, and even then its only positive contribution is for cases where there is no concurrency — the implementation described herein did not improve JVSTM’s scalability in any way.
CHAPTER 5. PROPOSED OPTIMIZATIONS

Figure 5.17: Performance of the adaptive IHS with adaptive validation with 1% write ratio

Figure 5.18: Performance of the adaptive IHS with adaptive validation with 50% write ratio
5.3 Collaborative Validation

5.3.1 Motivation

As described in Section 4.2.3, JVSTM employs a two-phase validation algorithm to guarantee a valid transaction commit order, where the first phase (snapshot validation) is mandatory and the second (incremental validation) is only necessary when two transactions try to enter the Active Transactions Record (ATR) simultaneously. These two phases are necessary because each transaction validates itself and then competes with others for a place in the ATR, which defines the commit order. Losing transactions then have to revalidate themselves against the one that entered the ATR and try again. The higher the number of transactions running simultaneously, the greater are the chances that two or more will be competing for a place in the ATR at the same time, and thus more time will be spent on incremental validations. Currently, a transaction’s lifecycle in JVSTM can be described as follows:

1. Execution
2. Help all transactions already in the ATR commit (collaborative commit)
3. Snapshot validation
4. Repeat incremental validation until successfully entering the ATR
5. Help write back/commit all transactions in the ATR up to, and including, itself.

With collaborative validation, I suggest that the snapshot validation phase is moved forward in the transaction’s lifecycle in such a way that incremental validations are rendered unnecessary without compromising the validity of the commit order, hopefully reducing the time it takes to validate a transaction under high concurrency. Also, instead of each transaction validating itself, all transactions that are not in the execution phase help the oldest uncommitted transaction validate using the snapshot algorithm.

To accomplish this, transactions are allowed to enter the ATR before being validated. The competition for a place in the ATR still exists (as previously, a CAS is used), but upon failure the transaction merely has to repeat the CAS and does not perform any kind of validation. Because of this change, the ATR now contains records of transactions that will later fail validation and abort, but it still represents the order in which valid transactions will be committed. In the example ATR shown in Figure 5.19, transactions $T_3$ and $T_5$ committed successfully. If transactions $T_6$, $T_7$ and $T_8$ are validated successfully, they will be committed in that order. If any of them fails validation, as happened with $T_4$, they will have no impact on the system despite the record being kept in the ATR. Instead, aborted transactions restart with a completely new record and enter the ATR at a later point.

Compared to the previous usage of the ATR shown in Figure 4.7, notice that records marked as “pending validation” are already inside the ATR and have also already been assigned a definite transaction number, that will only be used if these transactions end up being valid.

With these changes to the ATR and the validation process, a transaction’s lifecycle is now as follows:
CHAPTER 5. PROPOSED OPTIMIZATIONS

1. Execution

2. Enter the ATR

3. Help the oldest uncommitted transaction validate and commit (collaborative validation + collaborative write-back), repeat until the transaction is committed or aborted.

By moving validation forward in the transaction’s lifecycle and combining it with the collaborative write-back, as well as guaranteeing that no transaction in the ATR is validated before all previous transactions have either committed or aborted, a valid commit order is still guaranteed despite each transaction being validated only using snapshot validation.

Because this algorithm relies solely on snapshot validation, it can potentially reduce the overall cost of validating a transaction because the incremental process is no longer required.

5.3.2 Implementation

The original snapshot validation algorithm consists in going through the whole read set and verifying whether any box has been updated since the transaction started. Using the exact same approach in a collaborative version of the algorithm is not efficient as it would simply duplicate work amongst helping transactions.

Because of this, the ReadSet class was created. Similar to the WriteSet class that JVSTM already uses for the collaborative write-back, ReadSet divides a transaction’s read set into several blocks. While in WriteSet this division requires the write set’s HashMap to be transformed into two arrays, ReadSet reuses the arrays that the transaction used to store its read set during execution.

Recalling the example in Figure 5.19, let us assume $T_6$’s read set has 500 entries. With a block size of 50, the corresponding ReadSet will have 10 blocks. As previously described, all free transactions help the oldest uncommitted transaction validate and commit. In this example, it means $T_6$, $T_7$ and $T_8$ will help $T_6$ to validate, which they will do by starting at a randomly chosen block, as shown in Figure 5.20.

Having chosen a starting block, all boxes in that block are validated as they would be in snapshot validation, i.e. by checking whether they were updated after the validating transaction started. Each transaction then proceeds to the next block until all blocks have been validated. Like in WriteSet, each block has an AtomicBoolean variable associated to it that avoids duplicate work (when a transaction sees the block it is visiting is already marked as validated, it skips to the next block).

![Figure 5.19: Example of an ATR under collaborative validation](image-url)
5.4. LIMIT HELPERS

Any helping transaction only proceeds to the next phase (collaborative write-back) when it has observed that all blocks have been effectively validated either by itself or by other helping transactions, thus guaranteeing a valid commit.

5.3.3 Results

The experiments made showed that the idea of collaboratively validating transactions does not scale well. The reason for this is that where threads were previously spending their time validating themselves (which is guaranteed to be necessary work), with collaborative validation and under high concurrency there are too many threads trying to validate the same transaction, which results in unnecessary work being done, and thus higher execution times.

On the other hand, under low concurrency this solution at best matches the original implementation.

5.4 Limit Helpers

5.4.1 Motivation

As described in Section 4.3.4, JVSTM keeps a commit queue that defines the order in which valid transactions will be written back, and this process is done collaboratively, i.e. transactions help each other write back to minimize the amount of wasted time. There are two different points in a transaction’s lifecycle in which it can help others write back:

1. Immediately after exiting the execution phase and just before performing snapshot validation. This is done so that when the transaction is validated it observes the most recent view of world possible.

2. After entering the commit queue, the transaction helps write back all transactions that come before it, instead of waiting idle for older committing transactions to finish.

From this follows that all helping transactions will try to help the same transaction at the same time. Depending on the system’s workload, this algorithm can have the adverse effect of there being more helpers than a transaction really needs. Remember that a transaction’s write set is split into blocks for write back, which means that ideally there should not be more helpers than blocks to be written.
This section describes three methods of attempting to better manage how transactions perform the write-back phase — two of them by defining limits on the number of helpers, and another one that relaxes restrictions on the commit order.

5.4.2 Per-transaction Helper Limit

To limit the number of helpers on a per-transaction basis, an atomic counter must be added to each transaction’s record. Before starting to help, other helping transactions must first check whether the atomic counter of the transaction being committed falls below a certain threshold. This threshold is what will effectively limit the number of simultaneous helpers, and should always be lower than the transaction’s number of write set blocks, because otherwise some helpers are guaranteed to be redundant.

However, this leaves us with the question of what to do with excess helping transactions.

- “Wait and wait” strategy: the simplest solution is for all excess helpers’ threads to wait in idle until the system can proceed to write back the next transaction in queue, but this is not a promising proposition. Essentially, by doing this we will be moving from a scenario where some helpers do unnecessary work to a scenario where those helpers will do no work at all. In both scenarios excess helpers are not contributing in any way to move the system forward, and thus should have similar performance.

- The most intuitive course of action would be to help another transaction whose limit has not exceeded the threshold yet, but that requires additional changes to the JVTSM, because in the original version transactions must be written back sequentially one at a time. An implementation of those changes that allows the simultaneous write-back of different transactions is described in Section 5.4.3.

- A third approach is to decide what to do with an excess helper depending on which phase it is. Helpers that have yet to snapshot validate themselves can proceed directly to snapshot validation instead of helping queued transactions. While this is a way of keeping the system in motion without doing duplicate work, it also means that snapshot validation will be done earlier than before, and thus additional incremental validation steps may be needed. The balance between time saved by proceeding to snapshot validation and the extra time spend in incremental validation will dictate whether this idea is successful.

In turn, excess helpers that are already validated and in the commit queue can either be kept waiting (“snapshot and wait” strategy) or ignore the limit and help anyway (“snapshot and ignore” strategy). The latter is arguably the best choice, because there is a chance, even if small, that it will help the system move forward, whereas the former is guaranteed to do nothing.

5.4.2.1 Results

This idea was tested using different helper limits, but the results were roughly the same with all thresholds used. Thus, for brevity Table 5.6 shows the results for the different strategies using only one threshold (maximum of 2 helpers per transaction, except for the original implementation).
In all tests, each transaction’s write set has 100 boxes, which means that in the validation phase it will be divided into at most 10 blocks. Each transaction reads 900 boxes and the total number of boxes in the system is 1 million.

With 16 threads, all strategies have the same performance, as would be expected since with few simultaneous threads running there is less of a chance that too many threads are involved in the write-back process at once, and thus no room for improvement should be expected. Under high-concurrency, where better results were expected, it is unsurprising to see that both strategies that involve waiting in at least one of the helping phases do not offer visible improvements over the original algorithm. In turn, the “snapshot and ignore” strategy is 6.5% faster than the original JVSTM (23.2 s versus 24.8 s).

<table>
<thead>
<tr>
<th>Threads</th>
<th>Original</th>
<th>Wait and Wait</th>
<th>Snapshot and Wait</th>
<th>Snapshot and Ignore</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>23.5 s</td>
<td>23.5 s</td>
<td>23.5 s</td>
<td>23.4 s</td>
</tr>
<tr>
<td>96</td>
<td>24.8 s</td>
<td>24.6 s</td>
<td>24.5 s</td>
<td>23.2 s</td>
</tr>
</tbody>
</table>

Table 5.6: Comparison of different helper limit strategies

### 5.4.3 Simultaneous Write-back

As the name suggests, the idea of simultaneous write-back is to allow more than one transaction to be written back simultaneously, with the goal of reducing the potential for unnecessary work. However, this must be done keeping in mind that serialized write-back was necessary to guarantee that values were committed to the boxes in the correct order of versions. To comply with this requirement, the algorithm must check for write/write conflicts between transactions. A write/write conflict exists when two valid transactions’ write sets have at least one box in common, which means that the older transaction must be written back before the newer one can be committed (serial write-back). Otherwise, the order in which the transactions are committed is irrelevant, since they will not be committing to the same boxes.

![Figure 5.21: Example commit queue](image)
Figure 5.21 shows an example commit queue where $T_7$ was already committed, $T_8$ was already validated and is waiting to be written back, and transactions $T_{X..Y}$ are being validated. $T_8$’s write set contains boxes $A$ and $B$, $T_X$’s contains box $A$ and $T_Y$ wants to commit a value to box $C$. Let us assume there are no read/write conflicts between any of the transactions, meaning that both $T_X$ and $T_Y$ will eventually enter the queue.

To detect write/write conflicts, we will leverage the IdentityHashSet described in an earlier optimization to also keep track of boxes written. When $T_X$ wants to enter the queue, it will have to perform incremental validation against $T_8$ and at that point it will detect that box $A$ is in $T_8$’s IdentityHashSet, and thus will add that transaction’s record to its list of write conflicts. $T_X$ will then enter the queue as $T_9$ (Figure 5.22), $T_Y$ will validate itself against $T_{8..9}$ and enter the queue as $T_{10}$. This reduces the overhead of this technique.

![Figure 5.22: Example commit queue](image)

Under the original algorithm, all three transactions would be committed sequentially, one at a time, with the newer ones helping the older one. However, note that $T_{10}$ has no write conflicts, and thus can commit itself without waiting for older transactions (i.e. $T_{8..9}$). With the simultaneous write-back algorithm, $T_8$ and $T_{10}$ will start writing themselves back as soon as they enter the queue. A downside of this algorithm is that, despite $T_{10}$ being able to commit itself before $T_{8..9}$, the mostRecentCommittedRecord pointer can only move to $T_{10}$ after $T_{8..9}$ were committed, i.e. $T_{10}$’s results will not be made visible to new transactions immediately after $T_{10}$ finishes committing, i.e. the write set of a committed transaction is only visible once all previous transactions in the commit queue have committed, otherwise new transactions could observe an inconsistent view of the system.

$T_9$, which can only be written after $T_8$ has finished, can then do one of the following:

1. Keep waiting in a while loop until its conflicts have been resolved (“wait” strategy).

2. Help any other transaction that has no outstanding write conflicts. The simplest solution in this case would be to help the first transaction in the queue (“first” strategy), similarly to what happens in the original algorithm. Alternatively, one could choose to help the transaction with the least number of helpers.

3. Help the transactions with which it has a conflict (in this case, $T_8$) — “resolve” strategy.

The first solution is not ideal because it would result in having a thread effectively doing nothing until all conflicts have been resolved, which leaves us with the other two alternatives that
involve helping other transactions. The second option is similar to the original algorithm, and thus should have a neutral impact at best, whereas the last solution is the most promising: if we cannot write back a transaction because it has conflicts, the best course of action should be helping resolve those conflicts, so that the transaction can start writing itself back as soon as possible.

Note that these choices only concern the second helping phase (after the transaction enters the commit queue) and not the first one. Recall that the first phase occurs before a transaction is validated, and at that point it obviously cannot write itself back yet under any circumstances. Thus, that phase is left unchanged from the original algorithm, where transactions are committed sequentially.

5.4.3.1 Results

The idea for simultaneous write-back comes from a scenario where transactions are persisted not only to memory, but also to disk. In this scenario, the cost of writing to disk is high, and thus the possibility of having several transactions writing simultaneously can lead to performance gains. Transactional systems usually guarantee persistence to disk, and in this case this is simulated writing only in one file.

However, due to JVSTM’s help mechanism, where the same transaction can be written back by different threads, writing to disk would result in one the following problems:

- When writing back a transaction’s block, an helper thread would have to wait for the file to be closed by another thread that is also writing back the same transaction, or
- A transaction’s blocks would be scattered across different files.

To avoid those problems, a transaction’s write set must be written to disk in full by the same thread, which means JVSTM’s help feature cannot be used. Without further modifications, threads that would otherwise help a transaction commit must instead wait in idle until their own transaction can be committed. Obviously, this leads to the long execution times shown in Table 5.7 (under the “No Help” column).

In this context, simultaneous write-back emerges as an alternative to the write-back help algorithm. Although threads do not help each other commit, multiple threads can commit their own transactions simultaneously, and thus performance is much better.

<table>
<thead>
<tr>
<th>Threads</th>
<th>No Help</th>
<th>Sim. Write-back</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>48</td>
<td>127.5 s</td>
</tr>
<tr>
<td>Aborts</td>
<td></td>
<td>38.2 s</td>
</tr>
<tr>
<td>Time</td>
<td>96</td>
<td>2.702.729 s</td>
</tr>
<tr>
<td>Aborts</td>
<td></td>
<td>1.327.112 s</td>
</tr>
</tbody>
</table>

Table 5.7: Comparison of write-back to disk without help and with simultaneous write-back

However, it is also important to note that there is a significant increase both in time and aborted transactions when the number of concurrent threads is doubled (in both versions). A
possible reason for this in the simultaneous write-back version is that the disk becomes a bottleneck due to the fact that too many threads are writing to disk simultaneously. This problem is addressed in Section 5.4.4, but it is not the only reason because the same behavior happens in the version without help (where only one thread writes to disk at any given time).

To understand the rest of the problem, let us consider the following scenario:

- The system’s most recent version number is 1.
- Transaction $T_2$ is fully validated and ready to be written back.
- 10 new transactions are started.

Without write-back to disk, $T_2$ does not take much time to be written back to memory. Thus, there is a high chance that the new transactions being started already start with version number 2 and thus observe a newer version of the system, which reduces their chances of aborting during execution. On the other hand, when writing back values to disk, the write-back phase takes longer, and now the new transactions being launched are more likely to start with version number 1. Then, as soon as $T_2$ finishes write-back, any of those 10 transactions that tries to read a box changed by $T_2$ will abort. The higher the number of concurrent transactions, the greater becomes the probability of a transaction starting with a soon-to-be-changed view of the world and aborting. In turn, the increased number of aborts also results in longer execution times, and thus the difference in performance between 48 and 96 threads is explained.

To solve this problem, I changed the commit procedure from:

1. Write back to disk.
2. Write back to memory.
3. Mark the transaction as committed.

To the following:

1. Write back to memory.
2. Mark the transaction as committed.
3. Write back to disk.

With this change, transactions’ results are committed to memory and made available to new transactions much sooner, and the slow write-back to disk does not interfere with other transactions. Table 5.8 presents the results of this change. Note that when using this new order in a production environment, any failures that occur when writing to disk could lead to a rollback of the values written to memory, which would be problematic given that new transactions could already be working with those values.
5.5. TWEAK THE WRITE-BACK HELP ALGORITHM

<table>
<thead>
<tr>
<th>Threads</th>
<th>First to Disk</th>
<th>First to Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>48</td>
<td>38.2 s</td>
</tr>
<tr>
<td>Aborts</td>
<td>1.327112</td>
<td>628.516</td>
</tr>
<tr>
<td>Time</td>
<td>96</td>
<td>84 s</td>
</tr>
<tr>
<td>Aborts</td>
<td>11.491939</td>
<td>1.897284</td>
</tr>
</tbody>
</table>

Table 5.8: Comparison of the two different commit orders using simultaneous write-back

5.4.4 System-wide Helper Limit

Where the first idea limited the number of helper threads for each transaction, the goal here is to limit the total number of helper threads in the whole system, regardless of the number of transactions waiting to be written back. The main motivation for this idea came from the results observed in experiments where transactions’ results were also persisted to disk. In this experiment there is the possibility that having too many threads writing to disk at the same time leads to a bottleneck in the disk controller. When this happens, it is not enough to impose a limit of helpers per-transaction, and a system-wide limit is required. The implementation of such a limit is similar to the previously described (per-transaction) one, but instead of each transaction’s record having a counter, a system-wide counter is used. Because transactions are being written back to disk, the help algorithm is not used, and thus they proceed directly to snapshot validation. Whenever a transaction wants to do write-back work, it checks if that counter is lower than a pre-determined threshold. If the limit exceeds the threshold, the transaction waits for other transactions to finish writing. The ideal threshold will depend on each hard drive’s writing capabilities and for testing purposes will be chosen by trial and error.

5.4.4.1 Results

Testing with a maximum of 10 simultaneous threads writing to disk, in addition to the “First to Memory” simultaneous write-back order described in the previous section, lead to a reduction from 41.2 s to 36.2 s, with no impact on aborts.

5.5 Tweak the Write-back Help Algorithm

5.5.1 Motivation

When a transaction enters the validation phase, its write set is rebuilt into a new structure (the WriteSet class) that divides it into blocks of 10 boxes, so that it can be written back by multiple threads simultaneously.

In the original help algorithm (Listing 5.2), each helper starts at a random block. Then, for each block it checks whether an AtomicBoolean (blocksDone) flag is set and if not, it writes back that block to memory; otherwise, it proceeds to the next block. This process is repeated until all blocks are seen as committed. However, note that each block’s flag is only set after
it has effectively been written back to memory. This is needed to guarantee that if an helper thread stops for an unforeseen reason while it is in the middle of writing back a block, that block is seen as uncommitted by another helper and eventually written back.

This approach has the disadvantage that while a block is being written back, other helpers will not know that it is being written back already, which can lead to the same block being written back by more than one thread unnecessarily. The chances of this happening are mitigated because each thread starts helping in a random position, but this is not enough in scenarios where transactions have few blocks or where there are too many threads helping.

The following sections propose tweaks to this algorithm that aim to reduce duplicate work in the help algorithm. These tweaks do not apply to the simultaneous write-back algorithm, which is incompatible with the write-back help feature.

### 5.5.2 Use AtomicInteger Flags Instead of AtomicBoolean

While an AtomicBoolean flag only allows the algorithm to differentiate between two states — “not written back” (NWB) and “written back” (WB) —, the AtomicInteger class opens up the possibility to introduce an intermediate state — “write-back in process” (WBP). When a thread wants to write back a block, it attempts to perform a CAS to change that block’s status from NWB to WBP — if the CAS succeeds, it means that no other thread was already writing back that block. If the CAS fails, it means that block is already in a state other than NWB, and the thread proceeds to the next block. Each helper repeats this process until it observes all blocks in a state different than NWB.

However, this process alone does not offer the same safety guarantees as the original algorithm, because if a helper thread fails during write back of a block, no other thread will write it back. To prevent that from happening, all blocks must be checked again. In this second loop, even blocks that are in the WBP state will be written back. If the thread that marked a block as WBP is still writing it back, this will lead to duplicate work, but if it failed before finishing write-back, the algorithm (Listing 5.3) guarantees that the block is fully written back. Despite still allowing some duplicate work, that only happens in the second loop. Hopefully, by the time a thread reaches the second loop, most or all blocks will already have been marked as WB, and duplicate work will be reduced or eliminated.
protected final void helpWriteBack(int newTxNumber) {
    if (this.nBlocks == 0) return;

    int finalBlock = random.get().nextInt(this.nBlocks);
    int currentBlock = finalBlock;
    do {
        if (blocksDone[currentBlock].compareAndSet(0, 1)) {
            this.bodiesPerBlock[currentBlock] = writeBackBlock(currentBlock, newTxNumber);
            this.blocksDone[currentBlock].set(2);
        }
        currentBlock = (currentBlock + 1) % this.nBlocks;
    } while (currentBlock != finalBlock);

    do {
        if (blocksDone[currentBlock].get() != 2) {
            this.bodiesPerBlock[currentBlock] = writeBackBlock(currentBlock, newTxNumber);
            this.blocksDone[currentBlock].set(2);
        }
        currentBlock = (currentBlock + 1) % this.nBlocks;
    } while (currentBlock != finalBlock);
}

Code Listing 5.3: Write-back help algorithm with AtomicInteger status flags

5.5.3 Use AtomicInteger Flags and a Global AtomicBoolean Flag

The previous algorithm can be tweaked further by using a global boolean variable to mark whether any thread has observed all blocks as fully written back (WB state). As Listing 5.4 shows, after the second loop the global flag (allBlocksDone) is set to true. Also, each iteration of the second loop checks whether the flag has been set and if so, the loop stops immediately.

With this, after the first helper reaches the end of the algorithm, all other helpers can stop safely knowing that at least one thread has observed all blocks in the WB state, thus further reducing unnecessary work, while still matching the safety guarantees of the original algorithm.
protected final void helpWriteBack(int newTxNumber) {
    if (this.nBlocks == 0) return;

    int finalBlock = random.getInt().nextInt(this.nBlocks);
    int currentBlock = finalBlock;
    do {
        if (blocksDone[currentBlock].compareAndSet(0, 1)) {
            this.bodiesPerBlock[currentBlock] = writeBackBlock(currentBlock, newTxNumber);
            this.blocksDone[currentBlock].set(2);
        }
        currentBlock = (currentBlock + 1) % this.nBlocks;
    } while (currentBlock != finalBlock);

    do {
        if (allBlocksDone.get() != 2) {
            this.bodiesPerBlock[currentBlock] = writeBackBlock(currentBlock, newTxNumber);
            this.blocksDone[currentBlock].set(2);
        }
        currentBlock = (currentBlock + 1) % this.nBlocks;
    } while (currentBlock != finalBlock);
    allBlocksDone.set(true);
}

Code Listing 5.4: Write-back help algorithm with AtomicInteger status flags and global AtomicBoolean flag

5.5.4 Results

Table 5.9 shows the performance of JVSTM with the original write-back help algorithm and the two tweaked alternatives in a scenario where write operations represent 10% of the total number of operations. In these experiments, the only other optimization used is the hybrid read set (i.e. transactions are not written back to disk and there is no simultaneous write-back). Given the nature of the motivation for these tweaks to the algorithm, any improvements they bring should be more visible in tests with more concurrency, because in that case there are more threads running, and thus there is a greater chance of threads performing unnecessary work. Comparing the results of 16 and 96 threads shows that that is the case, but percentage-wise the improvements are at best 6.8%, which leads to the conclusion that this algorithm has little room for improvement, i.e. it already keeps unnecessary work at a minimum. Also of note is that the version which combines per-block AtomicInteger flags with a global AtomicBoolean flag has worse performance than the one without the global flag, probably because the amount of duplicate work it eventually saves is smaller than the time required to check whether the flag is set (which is done in each iteration of the second loop).
5.6 Upgrade Transactions

5.6.1 Motivation

In the JVSTM, when a transaction attempts to read a box whose version number is greater than the transaction’s own version number, it is aborted because this means that there is a conflict with another transaction that committed a new value to that box after the transaction started. The consequence of this is that the work the transaction performed before aborting is thrown away and it will have to restart from the beginning with a new version number.

A possible solution to avoid this waste of work when this occurs is to try to upgrade the transaction to a new version number first, instead of immediately aborting it. The rationale behind this idea is that if the transaction was running with a version number higher than the box’s, there would not be a conflict. If the upgrade succeeds, the transaction can continue normally and its prior work will not be wasted.

Naturally, we must guarantee that the transaction remains valid after it is upgraded. A transaction’s ability to be upgraded depends on whether the work it has performed until the upgrade is still valid under the transaction’s new version number. In other words, we must guarantee that the transaction’s current state (read and write sets) would be the same if the transaction had run with the new version number.

Since the transaction’s write set is only effectively written after the execution phase, changing the transaction’s version number during its execution does not have a direct effect in the transaction’s validity, and thus the write set can always be kept intact during an upgrade.

On the other hand, the read set is dependent on the transaction’s version number. When we upgrade a transaction from version $X$ to version $Z$ (where $X < Z$), it is as if it started later than it actually did, and thus its current read set is only valid if none of the boxes it contains was changed between $t_X$ and $t_Z$. In practice, this means that to upgrade a transaction we must run the snapshot validation algorithm on its current read set.

It follows that when a transaction $T_X$ attempts to read from a box $B$ whose most recent version $Y$ is greater than $X$, instead of aborting right away, JVSTM does the following:

1. A local reference to the current mostRecentCommittedRecord, whose version is $Z$ ($X < Y \leq Z$), is recorded.
2. Partial snapshot validation: if any of the boxes in $T$’s read set (which at this point does not contain the conflicting box $B$ yet) has a version number higher than $X$, the transaction cannot be upgraded and must be aborted.
3. If all the boxes $T$ already read remain valid, it can be upgraded to version $Z$.
4. The transaction can now continue reading $B$.

Note that the first step is needed because the `mostRecentCommittedRecord` can change if another transaction commits while we are verifying $T$. If that transaction happens to change a box we have already verified, without the local reference we would be mistakenly upgrading $T$ to a version where its read set is not really valid. By having a local reference that does not change during verification, if this scenario arises it will be detected at the latest when the transaction is validated.

Naturally, this upgrade process will only be beneficial if the time it saves (namely the time the transaction already spent executing until a conflict arises) is greater than the time it takes to verify the read set’s validity. From this follows that the longer the transaction has already run, the more time we can potentially save.

To test this theory, the following tests include different degrees of artificially inflated execution times, which are achieved by putting the thread to sleep every time the transaction writes a value. In practice, this simulates the execution of non-transactional operations, which a realistic application is likely to include, but the benchmark in use did not support (all operations in a transaction involved reading or writing shared boxes).

### 5.6.2 Results

<table>
<thead>
<tr>
<th>Time Inflation</th>
<th>No Upgrade</th>
<th>Upgrade</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>None</td>
<td>21.9 s</td>
<td>21.2 s</td>
</tr>
<tr>
<td></td>
<td>Aborts</td>
<td>789.231</td>
<td>546.220</td>
</tr>
<tr>
<td>Time</td>
<td>1 ms</td>
<td>1.828 s</td>
<td>1.378 s</td>
</tr>
<tr>
<td></td>
<td>Aborts</td>
<td>672.272</td>
<td>349.370</td>
</tr>
<tr>
<td>Time</td>
<td>2 ms</td>
<td>3.395 s</td>
<td>2.568 s</td>
</tr>
<tr>
<td></td>
<td>Aborts</td>
<td>670.554</td>
<td>350.026</td>
</tr>
</tbody>
</table>

Table 5.10: Comparison of the original JVSTM with and without upgrade (48 threads)

<table>
<thead>
<tr>
<th>Time Inflation</th>
<th>No Upgrade</th>
<th>Upgrade</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>None</td>
<td>24.1 s</td>
<td>21.5 s</td>
</tr>
<tr>
<td></td>
<td>Aborts</td>
<td>821.620</td>
<td>591.611</td>
</tr>
<tr>
<td>Time</td>
<td>1 ms</td>
<td>1.395 s</td>
<td>998 s</td>
</tr>
<tr>
<td></td>
<td>Aborts</td>
<td>1.223.685</td>
<td>631.715</td>
</tr>
<tr>
<td>Time</td>
<td>2 ms</td>
<td>2.564 s</td>
<td>1.824 s</td>
</tr>
<tr>
<td></td>
<td>Aborts</td>
<td>1.222.390</td>
<td>629.454</td>
</tr>
</tbody>
</table>

Table 5.11: Comparison of the original JVSTM with and without upgrade (96 threads)

Tables 5.10 and 5.11 compare JVSTM’s performance in its original implementation (i.e. with no other optimization) with and without the upgrade feature. The time inflation column refers to the time the thread is put to sleep every time its transaction makes a write operation to simulate non-transactional operations.
When transactions only operate on boxes (i.e. when we do not simulate other operations by putting the thread to sleep), upgrading transactions instead of aborting them results in a reduction of around 30% of the number of aborted transactions in both scenarios (48 and 96 threads). This only leads to negligible/modest gains in terms of execution time (3 and 11%) because the cost of upgrading transactions is only barely offset by the time this feature saves, especially with 48 threads. However, this allows us to conclude that even in a scenario where transactions only perform operations on shared boxes, the upgrade algorithm does not have a negative impact.

When we factor in additional, non-transactional operations, by simulating them with 1 and 2 ms sleep times, aborts are nearly halved and, more importantly, execution is at least 24% faster because this scenario has a better balance between upgrade cost and benefit.

### 5.7 Combination of Multiple Optimizations

This section presents the results obtained by combining multiple optimizations proposed in this chapter.

#### 5.7.1 Hybrid Read Set and Transaction Upgrade with Persistence to Disk

<table>
<thead>
<tr>
<th>Time Inflation</th>
<th>No Upgrade</th>
<th>Upgrade</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>None</td>
<td>39.9 s</td>
<td>36.9 s</td>
<td>8%</td>
</tr>
<tr>
<td>Aborts</td>
<td>1.401.821</td>
<td>831.596</td>
<td>41%</td>
</tr>
<tr>
<td>1 ms</td>
<td>1.855 s</td>
<td>1.404 s</td>
<td>24%</td>
</tr>
<tr>
<td>Aborts</td>
<td>672.378</td>
<td>351.494</td>
<td>48%</td>
</tr>
<tr>
<td>2 ms</td>
<td>3.430 s</td>
<td>2.594 s</td>
<td>24%</td>
</tr>
<tr>
<td>Aborts</td>
<td>672.681</td>
<td>351.184</td>
<td>48%</td>
</tr>
</tbody>
</table>

Table 5.12: Comparison of optimized JVSTM with and without upgrade (48 threads)

<table>
<thead>
<tr>
<th>Time Inflation</th>
<th>No Upgrade</th>
<th>Upgrade</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>None</td>
<td>82 s</td>
<td>87 s</td>
<td>−6%</td>
</tr>
<tr>
<td>Aborts</td>
<td>11.816.837</td>
<td>5.528.772</td>
<td>53%</td>
</tr>
<tr>
<td>1 ms</td>
<td>1.411 s</td>
<td>1.012 s</td>
<td>28%</td>
</tr>
<tr>
<td>Aborts</td>
<td>1.226.215</td>
<td>634.189</td>
<td>48%</td>
</tr>
<tr>
<td>2 ms</td>
<td>2.602 s</td>
<td>1.859 s</td>
<td>29%</td>
</tr>
<tr>
<td>Aborts</td>
<td>1.225.151</td>
<td>633.269</td>
<td>48%</td>
</tr>
</tbody>
</table>

Table 5.13: Comparison of optimized JVSTM with and without upgrade (96 threads)

Tables 5.12 and 5.13 compare JVSTM’s performance with and without the upgrade feature described in Section 5.6 in a scenario where the following features or optimizations are also present:
• Hybrid read set (traditional read set during execution and Identity Hash Set upon validation), as described in Section 5.1.3.

• Simultaneous write back (multiple non-conflicting transactions can be written back simultaneously), as described in Section 5.4.3.

• Write back to disk, which implies that each transaction is written back by the same thread, i.e. transactions do not help each other write back. Also, this means that the simultaneous write-back strategy is “wait”, i.e. transactions that cannot commit themselves must wait for their conflicts to be resolved.

The differences in performance with these optimizations and persistence to disk closely mirror those of the original version that were observed in Section 5.6.2: the best cost/benefit ratio of the upgrade algorithm is achieved when transactions’ execution times are longer. As was the case in the original version of the JVSTM, in this scenario the upgrade feature nearly halves the number of aborted transactions and reduces time by at least 24%.
Conclusions and Future Work

6.1 Conclusions

In Chapter 3 I started by studying a set of existing STMs to understand how they work, what problems their developers are faced with and the different approaches they have at their disposal to solve them. Not having any prior experience with STMs, the goal of this analysis was to introduce me to the key concepts of STMs.

Then, in Chapter 4 I studied JVSTM’s implementation in depth, which allowed me to understand how it works and thus identify the following potential areas for improvement, which were then addressed in Chapter 5:

1. Linear read set data structure.
2. Fixed validation process.
3. The validation process is not collaborative.
4. Duplicate work in the write-back help algorithm.
5. Conflicts always cause transactions to be aborted.

Section 5.1 approached the first problem by substituting the array used for transactions’ read sets with a non-linear data structure (the Identity Hash Set). While this solution proved effective for high concurrency scenarios, it did not perform as well as the original data structure in low concurrency scenarios. I then proposed a hybrid implementation that uses each structure at different times (the original array during execution and snapshot validation and the Identity Hash Set in incremental validation), that matches JVSTM’s prior results under low concurrency and performs better with high concurrency.

Another interesting conclusion of these experiments was that changing only one aspect of the STM (in this case the read set structure) can have interesting side effects. While the goal of changing the read set structure was only to save time in incremental validation, it also lead to a completely different abort distribution, with more transactions aborting earlier in the lifecycle.

The second problem was approached in Section 5.2 with a dynamic solution that only performs incremental validation under certain circumstances and reverts to the original two-phase process in other cases. Because this solution relies more on incremental validation, it performed poorly when implemented on the original JVSTM because it is incompatible with its linear read set implementation, which lead me to combine it with the first optimization that uses an Identity Hash Set. Unfortunately, the best results for this idea were observed in low concurrency scenarios, and thus it does not contribute to improve JVSTM’s scalability.
The third idea, described in Section 5.3 of applying the same collaborative principle of the write-back phase to the validation phase, also did not prove effective as it ends up creating new opportunities of the same work being done by two threads.

Also in the domain of duplicate work, Sections 5.4 and 5.5 discuss different methods of reducing duplicate work in the write-back help algorithm. The most relevant contribution in this domain is the simultaneous write-back algorithm described in Section 5.4.3, which also addresses the problem of persisting transaction data to disk.

Finally, Section 5.6 presented positive results of trying to save transactions that want to abort by upgrading them to a new version.

6.2 Future Work

Even though the benchmark used throughout this work is enough to compare the performance of different structures and algorithms and draw conclusions on their effect in JVSTM’s behavior, it does not represent a realistic usage scenario due to the following limitations:

- The shared object pool remains fixed throughout execution, whereas in a real environment transactions would create new boxes and delete existing ones, instead of simply updating the boxes that were created when the benchmark started.
- All transactions have a similar workload because they operate on the same number of boxes, whereas in a real application transactions are likely to have different write ratios and operate on different number of boxes.
- Because boxes are chosen completely at random, when a transaction aborts and restarts from scratch, its read and write sets will be completely different from before, whereas in a production environment transactions are much more likely to repeat the same tasks they wanted to do before.

Even though I have tried to address some of these and other issues in some tests, the ideas discussed in this document could have different results if they were to be tested using different benchmarks. Likewise, different testing scenarios could reveal potential optimizations that this work did not address.

In some experiments, I simulated persistence by writing to disk. Further work could improve this by using a database and experiment with different strategies to write data to the database.

Additionally, some of the algorithms presented in this work can be developed further to better perform in different environments.

As implemented, the helper limitation algorithms described in Section 5.4 assume hard coded thresholds that may not be suitable for all workloads. Future developments of such algorithms could include methods to dynamically set thresholds based on the current environment’s characteristics.

In several test scenarios, it often is the case that increasing the number of threads does not result in any time savings or actually leads to worse performance because under high concurrency the probability of aborts occurring increases. With this in mind, further experiments
where not all running threads are executing transactions could be done. For example, instead of having 96 threads running 96 transactions, JVSTM could have the same 96 threads, but only a portion of them would have an associated transaction, and the remaining threads would only be helpers, contributing exclusively to help the existing transactions. With this feature, there would be less transactions running simultaneously at any given time, thus leading to less conflicts, and the remaining processing power would help them finish faster. The success of this idea depends on the existence of a good balance between not executing too many transactions at once and not having too many threads helping.
Bibliography


package jvstm.util;

import java.util.Arrays;
import java.util.ConcurrentModificationException;
import java.util.Enumeration;
import java.util.NoSuchElementException;

public class IdentityHashSet<K> {
    private static final int DEFAULT_CAPACITY = 512;
    private static final float DEFAULT_LOAD_FACTOR = 0.75f;

    private int threshold;
    private final float loadFactor;
    private int size;
    private int mask;

    private static final class HashEntry<K> {
        HashEntry<K> next;
        K key;
        HashEntry(K key) {
            this.key = key;
        }
    }

    public IdentityHashSet() {
        this(DEFAULT_CAPACITY, DEFAULT_LOAD_FACTOR);
    }

    public IdentityHashSet(int initialCapacity) {
        this(initialCapacity, DEFAULT_LOAD_FACTOR);
    }

    public IdentityHashSet(int initialCapacity, float loadFactor) {
        buckets = (HashEntry<K>[][]) new HashEntry[initialCapacity];
        this.loadFactor = loadFactor;
        threshold = (int) (initialCapacity * loadFactor);
        mask = initialCapacity - 1;
    }

    public int size() {
        return size;
    }

    public boolean isEmpty() {
        return size == 0;
    }

    public Enumeration<K> keys() {
        return new KeyEnumerator();
    }

    public boolean containsKey(Object key) {
        int idx = key.hashCode() & mask;
HashEntry<K> e = buckets[idx];

while (e != null) {
    if (e.key == key)
        return true;
    e = e.next;
}
return false;

public boolean put(K key) {
    int idx = key.hashCode() & mask;
    HashEntry<K> e = buckets[idx];
    while (e != null) {
        if (e.key == key)
            return true;
        e = e.next;
    }

    // At this point, we know we need to add a new entry.
    if (++size > threshold) {
        rehash();
        // Need a new hash value to suit the bigger table.
        idx = key.hashCode() & mask;
    }
    e = new HashEntry<K>(key);
    e.next = buckets[idx];
buckets[idx] = e;
    return false;
}

public void clear() {
    Arrays.fill(buckets, null);
    size = 0;
}

private void rehash() {
    HashEntry<K>[] oldBuckets = buckets;

    int newcapacity = (buckets.length * 2);
    mask = newcapacity - 1;
    threshold = (int) (newcapacity * loadFactor);
    buckets = (HashEntry<K>[]) new HashEntry[newcapacity];
    for (int i = oldBuckets.length - 1; i >= 0; i--) {
        HashEntry<K> e = oldBuckets[i];
        while (e != null) {
            int idx = e.key.hashCode() & mask;
            HashEntry<K> dest = buckets[idx];

            if (dest != null) {
                HashEntry next = dest.next;
                while (next != null) {
                    dest = next;
                    next = dest.next;
                }
                dest.next = e;
            } else {
                buckets[idx] = e;
            }
        }
    }
    HashEntry<K> next = e.next;
    e.next = null;
    e = next;
private class EntryEnumerator implements Enumeration<HashEntry<K>> {
    /** The number of elements remaining to be returned by next(). */
    int count = size;
    /** Current index in the physical hash table. */
    int idx = buckets.length;
    /** Entry which will be returned by the next nextElement() call. It is set if we are iterating through a bucket with multiple entries, or null if we must look in the next bucket. */
    HashEntry<K> next;
    /** Construct the enumeration. */
    EntryEnumerator() {
        // Nothing to do here.
    }
    /** Checks whether more elements remain in the enumeration.
     * @return true if nextElement() will not fail. */
    public boolean hasMoreElements() {
        return count > 0;
    }
    /** Returns the next element.
     * @return the next element
     * @throws NoSuchElementException if there is none. */
    public HashEntry<K> nextElement() {
        if (count == 0)
            throw new NoSuchElementException("Hashtable Enumerator");
        count--;
        HashEntry<K> e = next;
        while (e == null)
            if (idx <= 0)
                return null;
            else
                e = buckets[--idx];
        next = e.next;
        return e;
    }
}

private final class KeyEnumerator implements Enumeration<K> {
    /** This entry enumerator is used for most operations. Only <code>nextElement()</code> gives a different result, by returning just the key rather than the whole element. */
    private EntryEnumerator enumerator;
    /** Construct a new KeyEnumerator */
    KeyEnumerator() {
        enumerator = new EntryEnumerator();
    }
    /** Returns true if the entry enumerator has more elements.
     * @return true if there are more elements
     * @throws ConcurrentModificationException if the hashtable was modified */
    public boolean hasMoreElements() {
        return enumerator.hasMoreElements();
    }
}
/** Returns the next element. 
 * 
 * @return the next element 
 * @throws NoSuchElementException 
 * if there is none. */ 
 public K nextElement() { 
     HashEntry<K> entry = (HashEntry<K>) enumerator.nextElement(); 
     K retVal = null; 
     if (entry != null) 
         retVal = entry.key; 
     return retVal; 
 }